
Ville Friman

# Testauslähtöinen ohjelmistokehitys

Metropolia Ammattikorkeakoulu
Insinööri(AMK)
Tietotekniikka
Insinöörityö
27.03.2013

Metropolia

## Tiivistelmä

| | |
|---|---|
| Tekijä | Ville Friman |
| Otsikko | Testauslähtöinen ohjelmistokehitys |
| Sivumäärä | 43 + 20 |
| Päivämäärä | 27.03.2013 |
| Koulutusohjelma | Tietotekniikka |
| Suuntautumisvaihtoehto | Ohjelmistotekniikka |
| Työn valvoja | Tomi Korpela |
| Työn ohjaaja | Auvo Häkkinen |


Insinöörityössä esitellään testauslähtöisen ohjelmistokehityksen vaiheet ja miten ne toimivat yhdessä.

Teoriaosuudessa käydään ensin läpi perinteinen V-mallin ohjelmistokehitys ja miten siinä ohjelmistotestaus on toteutettu. Testauslähtöisen ohjelmistokehityksen osiossa esitellään lyhyesti, miten käyttäjien haluamat ominaisuudet muutetaan käyttäjätarinoiksi, ja tarinat edelleen testeiksi.

Esimerkkiprojektissa tehdään ohjelma, joka hakee säätietoja verkkopalvelusta, muokkaa niitä ja tallentaa ne tietokantaan. Projektissa käytetään teoriaosuudessa esiteltyjä asioita ja miten niitä sovelletaan käytännössä. Projektin aluksi tehdään käyttäjätarina, joka muutetaan hyväksymistestiksi koko ohjelmalle. Lopuissa luvuissa esitellään, miten ohjelma rakennetaan paloissa kirjoittaen testit ensin.



**Avainsanat:** ATDD, TDD, Ohjelmistotestaus

Metropolia

## Abstract



| | |
|---|---|
| Author | Ville Friman |
| Title | Testauslähtöinen ohjelmistokehitys |
| Number of pages | 43 + 20 |
| Date | 27.03.2013 |
| Degree programme | Information Technology |
| Specialisation option | Software engineering |
| Supervisor | Tomi Korpela |
| Instructor | Auvo Häkkinen |




This bachelor's thesis presents how software projects are developed writing tests first. The theory part of the study explains first the traditional V-model and how testing is done. In the test driven development section it is shown how the customer's needs are turned into stories and those stories into user acceptance test cases. The sample project shows how test driven development, presented in the theory part, works in real a project. The project is about collecting weather information data and saving that into a database. The project shows how user stories are automated into tests and how the software is built piece by piece with test driven development



**Keywords:** ATDD, Software testing, TDD

# Sisältö

## Sanasto

**Acceptance Test Driven Development** Acceptance Test Driven Development Hyväksymistestauslähtöinen ohjelmistokehitys.

**eXtensible Hypertext Markup Language** HTML:stä kehitetty merkintäkieli, joka täyttää XML-kielen muotovaatimukset.

**eXtreme Programming** Eräs ketterän ohjelmistokehityksen projektinhallinta kehys.

**JavaScript Object Notation** JavaScript Object Notation Yksinkertainen tiedonsiirtomuoto.

**Test Driven Development** Test Driven Development Testauslähtöinen ohjelmistokehitys.

**Unified Modeling Language** Graafinen mallinnuskieli.

# 1 Johdanto

Tässä insinöörityössä esitellään miten ohjelmistoprojekti toteutetaan testauslähtöisesti ja mitä vaiheita prosessiin kuuluu. Tavoitteena on esitellä kehityksen vaiheet ja esimerkin avulla näyttää, miten niitä voidaan käyttää yhdessä.

Testauslähtöinen ohjelmistokehityksen ytimessä on asiakkaan tai käyttäjän tarpeiden ymmärtäminen. Näistä tarpeista johdetaan käyttäjätarinat, jotka muutetaan hyväksymistesteiksi. Hyväksymistestien jälkeen, ohjelmistoa aletaan rakentamaan pala kerrallaan testauslähtöisesti. Näin ohjelmistolle saadaan kattava testiverkko, joka toimii myös dokumentaationa ohjelmiston toiminnalle. Tämä testiverkko auttaa ohjelmiston jatkokehitystä, vikojen korjausta ja uusien ominaisuuksien lisäämistä.

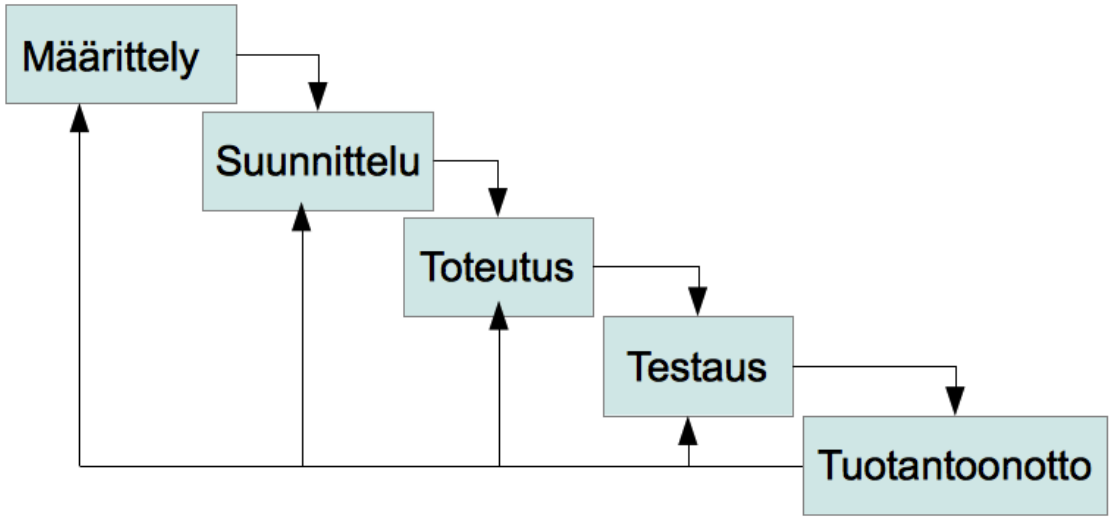
Insinöörityön esimerkkiprojektina toimii pieni ohjelma, jonka tarkoituksena on hakea säätietoja verkosta ja tallentaa niitä. Esimerkkiprojektissa käydään konkreettisesti läpi, miten teoriaosuudessa esitellyt vaiheet toimivat ohjelmistoprojektissa. Projekti alkaa ohjelman vaatimusmäärittelyllä, joista johdetaan ohjelmalle hyväksymistestit, käyttäen testauskehystä. Ohjelman toteutus käydään läpi tekemällä toiminnoille ensin yksikkötestit, ja sen jälkeen toiminnon toteutus.

# 2 Ohjelmistoprojektien testaus

Tietotekniikka on levinnyt nykyään lähes kaikkialle ihmisten ympärille. Nykyaikaiset asunnot ovat täynnä tietotekniikka, ihmiset asioivat verkkopankissa entisen konttorissa käynnin sijaan ja pitävät yhteyttä tuttaviinsa sosiaalisen median kuten Facebookin kautta. Samaan aikaan kun tietotekniikka on levinnyt kaikkialle, ovat ne myös monimutkaistuneet ja laajenneet. Tämä prosessi on myös vaikeuttanut ohjelmistojen testausta, ja siksi ohjelmistojen testaus on aina kompromissi käytössä olevien resurssien ja luotettavuudesta saavutetun varmuuden välillä (Haikala 2006, 238).

## 2.1 Testaus vesiputousmallissa

Vesiputousmalli (*Vesiputousmalli* 2012) on perinteinen ohjelmistoprosessi, joka on vielä käytössä. Kuvassa 1 on esitetty perinteisen vesiputousmallin vuokaavio.



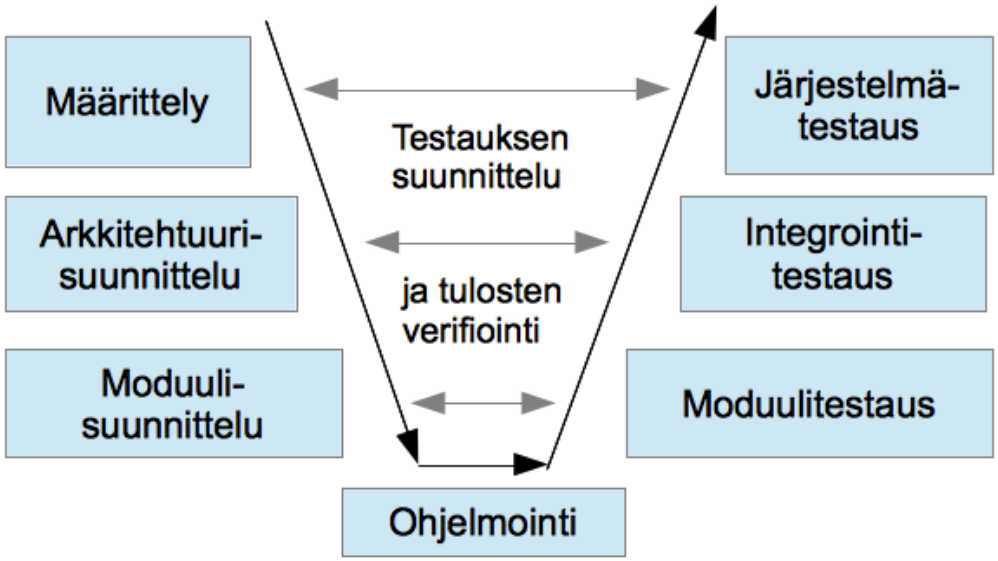

Kuva 1 Vesiputousmallin vuokaavio

Vesiputousmallin ensimmäinen vaihe on ohjemiston määrittelyiden kerääminen analysoiminen ohjelmiston toiminnallisten vaatimusten määrittelemiseksi. Määrittelyvaihe tuottaa ohjelmiston dokumentaation ohjelmiston toiminnallisista määrityksistä.

Suunnitteluvaiheessa suunnitellaan ohjelmiston arkkitehtuuri, josta saadaan ohjelmiston tekninen määrittely. Suunnitteluvaiheessa ohjelmisto myös pilkotaan mahdollisimman itsenäisiin osiin eli niin kutsuttuihin moduuleihin. Määrittelyn ja suunnitteluvaiheen ero on, että määrittelyvaiheessa kuvataan, mitä ohjelmisto tekee, ja suunnitteluvaiheessa miten se tehdään (Haikala 2006, s. 40).

Määrittely- ja suunnitteluvaiheiden jälkeen on katselmointi, jossa käydään vaiheen tavoitteet läpi ja tarkistetaan, että ne ovat täyttyneet. Katselmoinnissa tarkistetaan myös, että vaihe on tuottanut tarvittavat dokumentit.

Toteutusvaiheessa ohjelmisto tehdään määrittelyjen pohjalta. Testausvaiheeseen siirrytään, kun ohjelmistosta on ensimmäinen virheettömästi kääntyvä versio. Testausvaihe on tapahtuu yleensä monella, ja on V-mallin mukainen kuva 2 (Haikala 2006, s. 289).




Kuva 2 V-malli

Moduulitestauksessa testaan, että moduulin toiminta vastaa suunnitteluvaiheessa tehtyä teknistä määrittelydokumenttia, ja testauksesta vastaa yleensä

moduulin toteuttaja (Haikala 2006, s. 289). Integraatiotestauksessa testataan, että moduulit pystyvät toimimaan muiden kanssa ryhmässä. Samalla testataan, että moduulien rajapinnat toimivat. Testauksen tuloksia verrataan tekniseen määrittelydokumenttiin. Moduulitestaus ja integraatiotestaus tapahtuu yleensä rinnakkain. Järjestelmätestauksessa testataan koko järjestelmää ja sitä, että se vastaa määrittelyvaihessa tehtyyn dokumentaatioon. Järjestelmätestauksessa tehdään myös mahdolliset kuormitustestaukset, luotettavuustestaukset ja käytettävyystestaukset.

Testausvaiheen jälkeen ohjelmisto siirtyy tuotantoonottovaiheeseen, jossa se siirretään tuotantoon ja otetaan käyttöön.

## 2.2 Testaus ketterissä malleissa

Ohjelmistoalalla on noussut viimeisen vuosikymmenen aikana suosituksi kehistysprosesseiksi niin kutsutut ketterät menenetelmät, kuten XP (eXtreme Programming) ja Scrum. Ketterissä ohjelmistoprosesseissa ohjelmistokehitys jaetaan pieniin, yleensä kahden viikon mittaisiin paloihin, "sprintteihin". Sprinttien aikana on tarkoitus tuottaa jokin toimiva ohjelmiston osa. Ketterissä ohjelmistoproseissa testaus on koko ajan läsnä, eikä vain erillinen osa, kuten vesiputousmallissa.

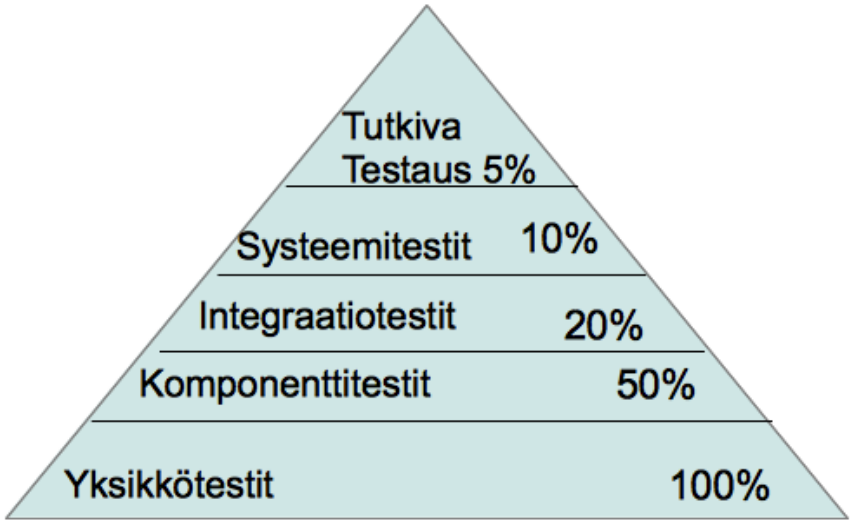
Yleensä ketterissä menetelmissä suositellaan käytettäviksi testivetoista ohjelmistokehitytä. Ketterissä ohjelmistomenetelmissä painotetaan huomattavasti testejä ja niistä saatavaa palautetta. Tätä varten on kehitetty jatkuvia integraatio-ohjelmistoja (continuous integration), jotka ajavat testejä esimerkiksi joka kerta, kun versionhallinnassa oleva lähdekoodi päivittyy.

Ketterissä menetelmissä panostetaan asiakkaan läsnäoloon projektissa. Asiakkaan kanssa suunnitellaan hyväksymistestit ominaisuuksille ja asiakkaalle järjestetään esittelytilaisuuksia projektin etenemisestä ja uusista ominaisuuksista. Esittelytilaisuuksia pidetään säännöllisin väliajoin, kuten kahden viikon

välein.

## 2.3 Testauksen tasot

Testauksessa on eri tasoja riippuen siitä, mitä testataan. Kuvassa 3 (Martin 2011, s. 115) on esitetty pyramidi, joka kuvaa testauksen eri tasoja, niiden suhdetta toisiinsa ja suositeltua testikattavuutta.




Kuva 3 Testauspyramidi kattavuusprosentteineen

Pyramidin pohjan muodostavat yksikkötestit, jotka testaavat yksittäisten luokkien, metodien tai funktioiden toimintaan. Seuraavana on komponenttitestit, jotka testaavat jotain laajempaa toiminnallisuutta. Pyramidin tasoja noustessa testien abstraktiotaso nousee ja niiden testikattavuus pienenee. Testikattavuus pienenee siirryttäessä kohti pyramidin huippua, koska ylempien tasojen ei tarvitse testata ja verifioida alemmilla tasoilla testattuja asioita. Esimerkiksi komponenttitestit voivat testata toiminnallisuutta päästä päähän välittämättä siitä, mitä ohjelma tai toiminto tekee välissä, koska yksikkötestit testaavat välissä olevat toiminnot. Ylemmillä tasoilla myös testien, jotka testaavat kulmatapauksia, määrä vähenee, koska ne on jo testattu alemmilla tasoilla.

Pyramidin tasot kuvaavat abstraktion lisäksi myös testien suorittamiseen kuluvaan aikaa. Yksikkötestien tulisi olla mahdollisimman nopeasti ajautuvia, koska niiden kattavuus on suurin ja niitä ajetaan usein. Tutkiva testaus on taasen ihmisen tekemään testausta, joka on hidasta eikä sitä tehdä usein. Seuraavissa luvuissa esitellään testauksen tasot hieman tarkemmin.

### 2.3.1 Yksikkötestit

Yksikkötestit muodostavat testauspyramidin pohjan. Yksikkötestien ominaisuuksia ovat nopea suoritusaika ja riippumattomuus ulkoisista järjestelmistä, kuten esimerkiksi tietokannasta tai web service -palveluista. Yksikkötestit toimivat myös dokumentaationa ohjelmiston seuraaville kehittäjille, sillä ne kuvavaat, miten ohjelmiston tulisi toimia (Martin 2011, s. 116). Esimerkissä 1 on listattu yksinkertaista luokkaa varten tehdyt yksikkötestit, ja testit läpäisevä luokka on listattu esimerkissä 2.

```
1   @Test
2   public void testParsingWithFirstNameLastName() throws Exception {
3           FullName fullName = parser.parse("Matti Mainio");
4           assertEquals("Matti", fullName.getFirstName());
5           assertEquals("Mainio", fullName.getLastName());
6   }
7
8   @Test(expected = IllegalArgumentException.class)
9   public void nullValueShouldThrowException() throws Exception {
10          parser.parse(null);
11  }
12
13  @Test(expected = IllegalArgumentException.class)
14  public void emptyStringShouldThrowException() throws Exception {
15          parser.parse("");
16  }
17
18  @Test(expected = IllegalArgumentException.class)
19  public void whiteSpacesShouldThrowException() throws Exception{
20          parser.parse("   ");
21  }
22
23  @Test(expected = IllegalArgumentException.class)
24  public void tooShortNameShouldThrowException() throws Exception{
25          parser.parse("Matti");
26  }
27
28  @Test(expected = IllegalArgumentException.class)
29  public void tooLongNameShouldThrowException() throws Exception{
30          parser.parse("Matti Jussi Mainio");
```

```
31          }
```

Esimerkki 1 Yksikkötestit yksinkertaiselle parsijalle.

```
1  package com.ville.friman.thesis;
2
3  public class NameParser {
4      private static final int FIRST_NAME = 0;
5      private static final int LAST_NAME = 1;
6
7      public FullName parse(String fullName) {
8          String[] splitNames = splitAndValidate(fullName);
9          return new FullName(splitNames[FIRST_NAME], splitNames[LAST_NAME]);
10     }
11
12     private String[] splitAndValidate(String fullName) {
13         checkForNullOrEmpty(fullName);
14         String[] splitNames = fullName.split(" ");
15         if(splitNames.length != 2){
16             throw new IllegalArgumentException("Parameter is too long or short");
17         }
18         return splitNames;
19     }
20
21     private void checkForNullOrEmpty(String fullName) {
22         if(fullName == null || fullName.trim().length() == 0){
23             throw new IllegalArgumentException("Parameter was null or empty");
24         }
25     }
26
27 }
```

Esimerkki 2 Testit läpäisevä parsija.

Yksikkötestien testikattavuus tulisi olla mahdollisimman lähellä sataa prosenttia. Täydellinen testikattavuus ei yleensä ole mahdollista eikä myöskään kannattavaa. Esimerkiksi Javassa aksessorien ja mutaattorien testaaminen ei ole järkevää, koska ne tulee testattua muiden testien yhteydesssä. Testikattavuus ei ole itseisarvo, vaan testikattavuuden tulisi kuvata todellista kattavuutta, jossa testattavan koodin toiminnallisuus myös verifioidaan.

### 2.3.2 Komponenttitestit

Tässä komponentilla tarkoitetaan luokkaa tai luokkia, jotka pitävät sisällään liiketoimintalogikkaa. Komponenttitestit voivat olla johdettu ohjelmiston hyväksymisvaatimuksista, jolloin komponenttitestit ovat ohjelmiston hyväksymistestit ja testaavat, että ohjelmisto toimii oikein. Komponenttitesteillä siis

testataan jonkin selkeän kokonaisuuden toimintaa. Niiden ei tule testata jokaista mahdollista testitapausta, vaan keskittyä varmistamaan ohjelmiston oikea toiminnallisuus. Yksikkötestit olivat dokumentaatio ohjelmistokehittäjältä toiselle ohjelmistokehittäjälle ja esittivät miten yksittäisen luokan olisi tarkoitus toimia. Komponenttitestit ovat dokumentaatio siitä, miten koko ohjelmiston tulisi toimia. Testipyramidin mukaan komponenttitestien osuus tulisi olla noin 50 %.

Komponenttitesteissä olisi suositeltavaa käyttää jotain hyväksymistestaukseen tarkoitettua ohjelmistokehystä, kuten esimerkiksi Concordion, Fittness tai Robot. Näiden ohjelmistokehyksien avulla esimerkiksi selkeiden raporttien tuottaminen liiketoiminnalle on helppoa. Kehykset eivät vaadi hyvää ohjelmointitaitoa, joten ohjelmoinnin perusteet hallitseva testaaja pystyy itse tekemään ja automatisoimaan komponenttitestejä.

### 2.3.3 Integraatiotestit

Integraatiotestit testaavat, että komponentit pystyvät yhteistyöhön. Integraatiotestit eivät siis testaa enää liiketoimintalogiikkaa vaan sitä, että ohjelmiston komponentit osaavat toimia yhdessä. Integraatiotestit eivät ole välttämättömiä, jos ohjelmisto koostuu vain muutamasta komponentista.

Komponenttien yhteistyön testaamisen lisäksi integraatiotesteissä voidaan ajaa suorityskykyä mittaavia testejä (Martin 2011, s. 118).

### 2.3.4 Järjestelmätestit

Järjestelmätestit ajetaan koko ohjemistoa vasten. Järjestelmätestit testaavat, että ohjelmisto toimii kokonaisuutena niin kuten se on suunniteltu. Liiketoimintalogiikkaa ei testata tällä tasolla testata, vaan liiketoimintalogiikka on testattu jo alemmilla tasoilla. Tällä tasolla ajetaan yleensä myös suorityskykyja kuormitustestit (Martin 2011, s. 118).

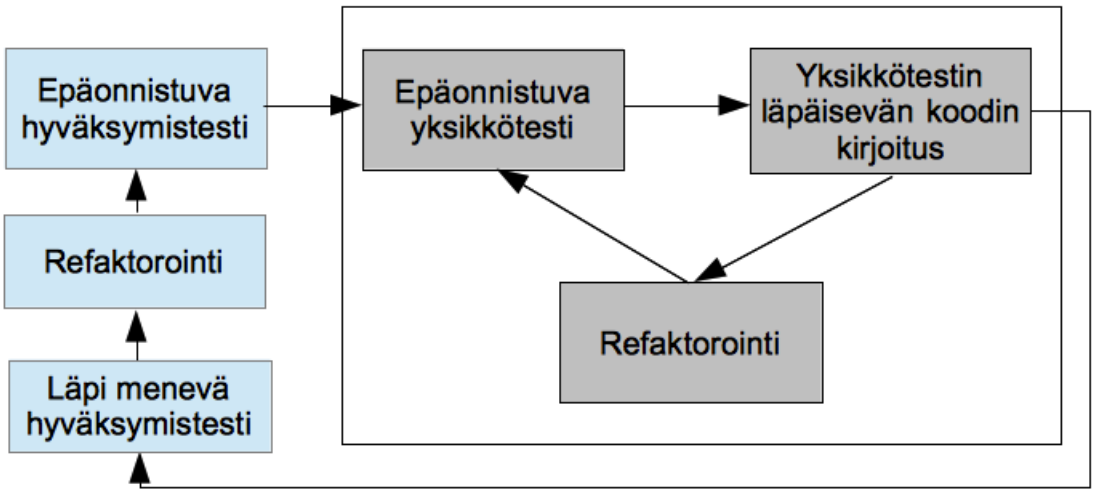
### 2.3.5 Tutkiva testaus

Tutkivassa testauksessa testaaja käy läpi ohjelmiston toimintaan käyttämällä ohjelmistoa. Testaajalla voi olla apunaan valmiiksi määriteltyjä testitapauksia, joita testaaja käy läpi ja merkitsee löydetyt virheet. Testaaja voi myös testata ohjelmistoa ilman etukäteen määriteltyjä testitapauksia, jolloin testaaja toimii niin kuin ohjelmiston tuleva käyttäjä. Tutkivassa testauksessa testaajan tarkoitus ei ole käyttää ohjelmaa satunnaisesti ja toivoa löytävänsä vikoja. Tutkivassa testauksessa testaajan tulisi hyödyntää tietojaan ohjemiston rakenteesta ja toiminnasta virheiden löytämiseksi.

Tutkivassa testauksessa ei ole tarkoituksena enää kasvattaa testikattavuutta, vaan testata, miten ohjelmisto toimii normaalissa tilanteessa, jossa ihminen on käyttäjä.

# 3 Ohjelmiston testauslähtöinen kehitys

Ohjelmiston testauslähtöinen kehitys pitää sisällään kaksi vaihetta, hyväksymistestauksen (ATDD, Acceptance Test Driven Development) ja yksikkötestauksen (TDD, Test Driven Development). Hyväksymistestausvaihe keskittyy määrittelemään ohjelmiston, tai sen osan, toiminnallisuuden. Yksikkötestausvaihe keskittyy toiminnallisuuden toteuttamiseen ohjelmistossa. ATDD:n ja TDD:n yhdistetyssä mallissa hyväksymistestit kertovat, mitä ominaisuutta ohjelmistokehittäjä alkaa tehdä ja milloin ominaisuus on valmis. Yksikkötestien avulla puolestaan varmistetaan, että ohjelmiston sisäinen rakenne tulee testatuksi ja pysyy selkeänä. Kehistystapojen yhdistetty vuokaavio on esitetty kuvassa 4




Kuva 4 ATDD:n ja TDD:n yhdistetty kehityssykli

Kuvassa ATDD:hen liittyvät osat ovat vaalean sinisellä pohjalla ja TDD:hen liittyvät ovat laatikon sisällä vaalean harmaalla. Kehitys aloitetaan kirjoittamalla ja automatisoimalla hyväksymistesti. Hyväksymistesti ei luonnollisestikaan mene läpi, koska toiminnallisuutta ei ole vielä kirjoitettu. Seuraavana on normaali TDD-sykli epäonnistuvan testin kirjoittamisen, testin läpäisevän koodin kirjoittamisen ja refaktorointivaiheineen. TDD-vaiheessa kirjoitetaan pala palalta hyväksymistestin läpäisevä toiminnallisuus. TDD-syklejä voi olla 1..N kappaletta, mutta kun hyväksymistesti menee läpi, tietää ohjelmistokehittäjä, että nyt voi lopettaa toiminnalisuuden kirjoittamisen. Kun hyväksymistesti menee läpi, niin voidaan suorittaa tarvittavat refaktoroinnit.

## 3.1 Hyväksymistestausvetoinen ohjelmistokehitys

Ohjelmiston käyttäjät ovat harvoin kiinnostuneet siitä, miten koodi toimii tai miten hienosti koodi on kirjoitettu. Käyttäjät ovat kiinnostuneet siitä, että ohjelmisto toimii oikein. ATDD:ssä pyritään varmistaan, että ohjelmisto toimii asiakkaan tai käyttäjän haluamalla tavalla. Ohjelmisto, joka toimii teknisesti täysin oikein, mutta ei tee sitä, mitä käyttäjä haluaa sen tekevän, on arvoton käyttäjälle.

### 3.1.1 Käyttäjätarinat

Halutun toiminnallisuuden määrittelemisessä käytetään yleensä käyttäjätarinoita. Käyttäjätarinat ovat lyhyitä kuvauksia ohjelman toiminnoista ja ominaisuuksista. Tarinat voivat olla vapaamuotoisia, mutta yksi suosituimmista tavoista on muoto: <rooli> haluan, että <toiminto>, jotta <hyöty> , esimerkiksi "Kirjautuneena käyttäjänä haluan, että lukemattomat viestini näkyvät etusivulle, jotta pääsen niihin helposti käsiksi". Taulukossa 1 on avattu esimerkkinä käytetty käyttäjätarina kahdeksi testiksi.

Taulukko 1 Käyttäjätarina avattu testitapauksiksi.

| **Tarina** | **Testitapaukset** |
|---|---|
| Kirjautuneena käyttäjänä haluan, että lukemattomat viestini näkyvät etusivulle, jotta pääsen niihin helposti käsiksi. | - Kirjaudu käyttäjänä ja varmista, että lukemattomien viestin määrä näkyy etusivulla.<br>- Jos käyttäjällä ei ole lukemattomia viestejä, näkyy teksti "Ei uusia viestejä". |

Kannatttaa huomata, että roolit eivät rajoitu pelkästään käyttäjiin, vaan voivat olla esimerkiksi teknisiä. Käyttäjätarinoiden avulla voidaan määritellä komponenttitason (kuva 3) testit, jolloin yksikkötesteissä keskitytään ohjelman sisäisen toiminnan testaamiseen ja käyttäjätarinoiden avulla testataan, että ohjelma toimii ulkoisesti kuten pitää.

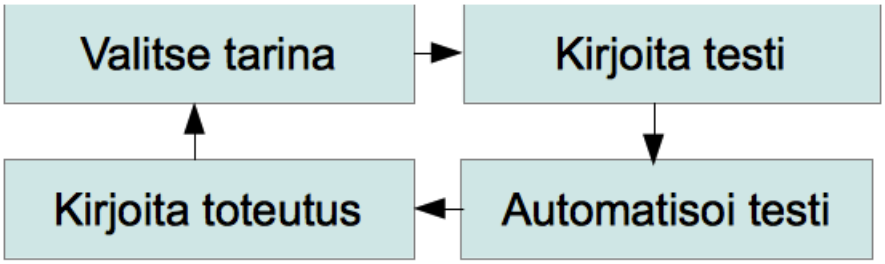
Alla on listattuna Lasse Koskelan (Koskela 2008, s. 328) määrittely hyväksymistesteille.

- asiakkaan omistama
- asiakkaan, kehittäjän ja testaajan yhdessä kirjoittama
- vastaa kysymykseen mitä, ei miten
- kirjoitettu ongelman toimialueen kielellä

- ytimekäs, tarkka ja yksiselitteinen.

### 3.1.2 Hyväksymistestausvetoisen kehityksen sykli

Hyväksymistestausvetoisen kehityksen sykli on esitetty kuvassa 5 (Koskela 2008, s. 335). Seuraavissa luvuissa esitellään sykli vaihe vaiheelta läpi.




Kuva 5 Hyväksymistestausvetoisen kehityksen sykli

### 3.1.3 Tarinan valinta

Ensimmäiseksi ohjelmistokehittäjä valitsee toteutettavan tarinan tai ominaisuuden. Tarinan valinnassa ei ole juurikaan ihmeellisyyksiä, sillä tarinat ovat yleensä jossain järjestyksessä, kuten tärkeysjärjestyksessä tai suositeltavassa toteutusjärjestyksessä. Ohjelmistokehittäjän näkökulmasta valintaan vaikuttaa myös, kuinka hyvin tarinan toimialueen osaa, sillä ei ole aina järkevintä valita sitä tarinaa, jonka osaa parhaiten toteuttaa. Jos ohjelmistokehittäjät tekevät aina vain tarinoita, jotka he voivat tai osaavat tehdä nopeasti, johtaa tämä nopeasti osaamisen henkilöitymiseen ja tiedon siiloutumiseen.

### 3.1.4 Testitapauksen kirjoittaminen

Aluksi tarinalle kirjoitetaan hyväksymistestitapaukset tuoteomistajan avustuksella. Tarkoituksena on tuottaa aineisto, joka sisältää testitapaukset, joiden avulla ohjelmiston oikea toiminta voidaan todentaa. Tapausten kirjoittamisessa tulisi mukana olla vähintään yksi henkilö, joka ymmärtää toimialueen ja pystyy sekä selittämään että tulkitsemaan tuoteomistajan toivomukset. Hyvä yhdistelmä on testaaja, jolla on toimialuetuntemusta ja joka pystyy tulkitsemaan tuoteomistajan vaatimukset, sekä ohjelmiston toteutuksessa muka-

na oleva ohjelmistokehittäjä. Ohjelmistokehittäjän rooli tässä on esittää tarkentavia kysymyksiä tuoteomistajalle. Kaikkien osanottajien tulee ymmärtää, mitä testitapaukset testaavat ja miksi. Kun testitapaukset on saatu kirjoitetuksi, on hyvä tarkistaa, että testitapauksissa ei ole päällekkäisyyksiä, ja poistaa ne.

#### 3.1.5 Testin kirjoitus

Kun ohjelmiston hyväksymistestitapaukset ovat selvillä, pitää ne saada seuraavaksi automatisoitua. Automatisoinnin voi tehdä testaaja, jos hänellä on tietotaitoa, tai ohjelmistokehittäjä. Testitapausten automatisoinnissa voidaan käyttää apuna esimerkiksi kappaleessa 2.3.2 mainittuja ohjelmistokehyksiä. Kun testi on automatisoitu, voi ohjelmistokehittäjä alkaa toteuttamaan ominaisuutta esimerkiksi TDD:n avulla. Ominaisuus on valmis, kun hyväksymistesti menee läpi.

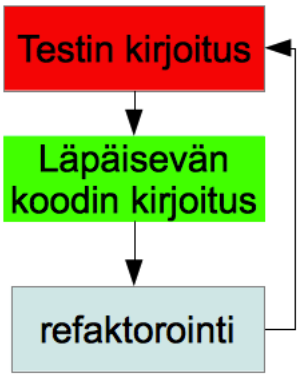
#### 3.1.6 Toiminnallisuuden toteutus

Tässä vaiheessa sykliä on tarinalle kirjoitettu hyväksymistesti tai -testejä, joille pitäisi tehdä toteutus. Ennen kuin toteutusta voidaan alkaa tehdä, tarina tulee pilkkoa tehtäviin. Tehtävät ovat yleensä pieniä tarinan osakokonaisuuksia, jotka voidaan tehdä enemmän tai vähemmän toisistaan riippumatta. Näitä tehtäviä aletaan yksi kerrallaan tehdä valmiiksi. Toteutustapa on vapaa, koska hyväksymistestien ei tulisi ottaa kantaa siihen, miten asia tehdään. Tehtävien tekotapa on yleensä testivetoinen kehitys, mutta tämä ei ole mitenkään pakollista.

### 3.2 Testauslähtöinen ohjelmistokehitys

TDD:llä tarkoitetaan toimintaa, tai sen osaa, testaavan testin kirjoittamista ennen varsinaisen lähdekoodin kirjoittamista. TDD on hyvin ohjelmistokehittäjälähtöinen tapa kehittää ohjelmistoja. Se keskittyy tuottamaan korkealaatuista koodia toimien samalla matalan tason dokumentaationa siitä, miten ohjelmisto toimii. Testivetoinen ohjelmistokehitys jakaantuu kolmeen osaan:

testin kirjoitukseen, läpäisevän koodin kirjoitukseen ja refaktorointiin (kuva 6 ).




Kuva 6 TDD-vuokaavio

Kuvan kahdessa ensimmäisessä vaiheessa olevat värit kuvaavat testin onnistumista: punainen testi ei mene läpi, vihreä testi menee läpi. Ohjelmistokehittäjän tulisi pitää kaikki osat mahdollisimman pieninä sekä ajallisesti, että koodirivien mukaan laskettuna. Kun osat pidetään mahdollisimman pieninä, on ohjelmistokehittäjän virheen sattuessa helpompi hylätä virheellinen koodi ja palata takaisin edelliseen toimivaan versioon koodista.

### 3.2.1 Testin kirjoitus

Punainen tarkoittaa testivietoisesssa ohjelmistokehityksessä epäonnistuvaa testiä. Ohjelmistokehittäjä kirjoittaa ensimmäiseksi koodin, joka testaa haluttua ominaisuutta. Kun testi kirjoitetaan ensin, joutuu ohjelmistokehittäjä miettimään ensiksi, mitä hän haluaa, eikä sitä miten hän haluaa sen (Steve Freeman 2009, s. 39). Jos ohjelmistokehittäjä ei pysty keksimään, mitä haluaa, niin ehkä haluttu ominaisuus ei ollutkaan selkeästi kuvattu. Kirjoittamalla testin ensin, ohjelmistokehittäjä joutuu miettimään toiminnallisuutta kutsujan kannalta. Testauslähtöisellä kehityksellä saadaan yleensä myös koodia, joka on sidottu mahdollisiin ulkoisiin riippuvuuksiin löyhästi. Tämä siksi, että koodin testaaminen muuttuu hyvin nopeasti hankalaksi, jos ohjelmistokehittäjä ei pysty korvaamaan ulkoisia riippuvuuksia omilla toteutuksillaan.

Henkilökohtaisessa kehityksessä olen myös huomannut, että kirjoittamalla testin ensin tulee ajatelleeksi enemmän erikoistapauksia ja esimerkiksi sitä, miten ohjelman tulee toimia viallisen syötteen kanssa.

### 3.2.2 Testin läpäisevän koodin kirjoitus

Punaisessa osassa kirjoitettiin testi, joka epäonnistuu. Vihreässä osassa kirjoitetaan koodi, joka saa testin menemään läpi. Tarkoituksena on kirjoittaa koodia vain sen verran, että testi menee läpi. Tämä saattaa alkuvaiheessa tarkoittaa, että toteutus palauttaa vakioita välittämättä metodille annetuista parametreista. Tässä vaiheessa ohjelmistokehittäjän ei tulisi välittää parametrien nimeämisistä, metodin rakenteesta tai pituudesta, vaan saada testi menemään läpi mahdollisimman nopeasti rikkomatta mahdollisia olemassa olevia testejä.

### 3.2.3 Refaktorointi

Syklin viimeinen osa on refaktorointi. Refaktoroinnilla tarkoitetaan koodin sisäisen rakenteen muuttamista muuttamatta ulkoista toimintaa (Fowler 2012). Refaktorointi on helpointa selittää esimerkin avulla. Esimerkki 3 on ensimmäinen versio kappaleessa 2.3.1 olevalle esimerkki 2:lle.

```
1  package com.ville.friman.thesis;
2
3  public class NameParser {
4
5          public FullName parse(String fullName) {
6                  if(fullName == null || fullName.trim().length() == 0){
7                          throw new IllegalArgumentException("Parameter was null or empty");
8                  }
9                  String[] splitNames = fullName.split(" ");
10                 if(splitNames.length != 2){
11                         throw new IllegalArgumentException("Parameter is too short");
12                 }
13                 FullName name = new FullName(splitNames[0], splitNames[1]);
14                 return name;
15         }
16 }
```

Esimerkki 3 Ensimmäinen versio NameParser.javasta

Esimerkki 3 läpäisee kaikki esimerkissä 1 esitellyt testit, joten ulkoiselta toiminnaltaan se on täysin vastaava kuin esimerkki 2. Koodin luettavuudessa on kuitenkin hyvin paljon eroa. Esimerkissä 3 luokan koko logiikka on yhden julkisen metodin sisällä ja tämän takia koodin kulkua on vaikea seurata. Esimerkissä 3 parametrin tarkistus ja jakaminen on siirretty omaan metodiinsa nimeltään splitAndValidate, jolloin päämetodiin jää jäljelle vain FullName-olion luominen ja palauttaminen. SplitAndValidate-metodista on vielä eriytetty tyhjän tai null-parametrin tarkistus omaksi metodikseen. Näin on saatu aikaiseksi yksinkertaisia selkeitä metodeja, jotka tekevät vain yhtä asiaa ja joiden toimintaa on helppo lukea.

Esimerkki 2 käytetään myös muuttujiksi muutettuja indeksejä, joilla on kuvaavat nimet, viittaamaan splitNames-listan alkioihin. Tämä selkeyttää, missä järjestyksessä nimet ovat listassa ja missä järjestyksessä ne annetaan FullName-luokalle.

Esimerkkinä käytetty luokka on hyvin yksinkertainen, mutta siitä saatiin hyvin pienillä refaktoroinneilla huomattavasti luettavampi verrattuna ensimmäiseen versioon. Esimerkin avulla on myös helppoa ymmärtää, miksi refaktorointi on hyvin tärkeää isommilla ja huomattavasti monimutkaisemmilla luokilla. Ilman refaktorointia ne muuttuvat hyvin epäselviksi ja niiden toimintalogiikan seuraaminen muuttuu vaikeaksi.

Refaktorointi ei tarkoita koko koodin kirjoittamista uudelleen, vaikka refaktoroinnin jälkeen alkuperäisestä koodista saattaa olla hyvin vähän jäljellä. Refaktorointia tulee tehdä niin tuotantokoodille kuin testikoodille. Näillä voi kuitenkin olla erilaiset rajat. Esimerkiksi testikoodissa voidaan hyväksyä koodin toistoa selkeyden vuoksi.

### 3.3 Testivetoinen ohjelmistokehitys olemassa olevassa koodissa

Testivetoinen ohjelmistokehityksen sykli voi muuttua, kun tehdään muutoksia olemassa olevaan koodiin. Jos koodilla on jo olemassa hyvät ja kattavat testit, niin sykli on sama kuin uuden koodin kanssa: testi, koodi ja refaktorointi.

Sykli kuitenkin muutttuu, jos olemassa olevalle koodille ei ole testejä, tai niitä on hyvin vähän. Jos testejä ei ole, tulisi ohjelmistokehittäjän kirjoittaa testejä olemassa olevalle toteutukselle, jotta säilyisi edes jonkinlainen varmuus, että olemassa olevaa toiminnallisuutta ei ole rikottu. Jotta testejä voisi tehdä, saattaa koodi vaatia refaktorointia, joka voi rikkoa olemassa olevaa toteutusta. Tähän muna-kana-ongelmaan löytyy apua nykyaikaisista kehitysympäristöistä, mutta toinen tapa on vaihtaa testauksen tasoa.

Sen sijaan, että ohjelmoija kirjoittaisi yksikkötestit olemassa olevalla koodille, hän kirjoittaa sille komponettitason testit. Tämä ei tietenkään korvaa yksikkötestejä, mutta on yleensä helpompi, nopeampi ja turvallisempi tapa saada koodille testejä. Kun komponenttitestit on kirjoitettu, voidaan koodia hieman luottavaisemmin alkaa refaktoroida.

# 4 Esimerkkiprojekti

Esimerkkiprojektissa toteutetaan pieni ohjelma ATDD:llä ja TDD:llä. Ohjelman tarkoitus on kutsua erästä avointa rajapintaa, josta se saa vastaukseksi JSON-muodossa (JavaScript Object Notation) alueen kaikkien kelikameroiden liikennesään.

Ohjelma käsittelee syötteen ja tallentaa sen tietokantaan myöhempää käyttöä varten. Ohjelma on osa vapaa-ajan projektia, jossa tarkoitus on visualisoida säätä.

Ohjelmalle ei kirjoiteta integraatio- ja systeemitestejä, koska ohjelma on yksinkertainen. Ohjelman toiminta kokonaisuutena voidaan helposti, nopeasti ja luotettavasti testata tutkivalla testauksella, eli ajamalla ohjelma ja tarkistamalla lopputulos.
Seuraavissa luvuissa käydään läpi projektissa käytettävät ohjelmistot ja ohjelman toteutus.

## 4.1 Projektissa käytettävät ohjelmistot

Tässä luvussa esitellään projektissa käytettävät ohjelmistot ja kirjastot. Kaikki ovat avoimen lähdekoodin ohjelmistoja ja ilmaiseksi saatavilla.

### 4.1.1 Staattiset analysaattorit

Staattisilla analysaattoreilla tarkoitetaan ohjelmistoja, jotka tutkivat ohjelmiston lähdekoodia mahdollisten virheiden, suorituskykyongelmien ja muiden vikojen löytämiseksi. Analysaattoreilla löydetään yleensä ongelmia, joita koneen on helppo havaita, mutta ihmisen ei. Tällaisia ovat esimerkiksi if-lause, joka ei koskaan toteudu, turhat muuttujat tai koodilohko, jota ei koskaan ajeta.

Staattisilla analysaattoreilla voidaan myös määrittää tyylitiedostoja, joihin

lähdekoodia verrataan. Tyylitiedostojen avulla pystytään varmistumaan siitä, että ohjelmiston koko lähdekoodi on kirjoitettu samalla tavalla. Tätä halutaan silloin, kun ohjelmisto on laaja ja sitä kehittää monta ohjelmistokehittäjää. Projektissa käytetään findbugsia (*Findbugs* 2012) ja PMD:tä (*PMD* 2012). Molemmat tekevät samoja asioita mutta hieman eri tavalla.

### 4.1.2 Testauskehykset

Testauskehykset on tarkoitettu helpottamaan testien kirjoittamista ja suorittamista. Ne tarjoavat valmiita metodeja syötteiden vertailuun ja metodien sisäisten toimintojen todentamiseen. Seuraavat kappaleet esittelevät lyhyesti projektissa käytetyt kehykset.

*JUnit*

JUnit (*JUnit* 2012) on eräs tunnetuimmista ja vanhimmista testikehyksistä Javalle. JUnit on osa niin kutsuttua xUnit-perhettä, joka tarjoaa yksikkötestikehyksen monelle ohjelmointikielelle. Ensimmäisen xUnit-kehyksen kehitti Kent Beck Smalltalk-ohjelmointikielelle. Javalle kehyksen siirsivät Kent Beck ja Erich Gamma (*XUnit* 2012). Projektissa JUnitia käytetään yksikkötestien kirjoittamiseen, ja Concordion-kehys vaatii sen toimiakseen.

*Concordion*

Concordion on hyväksymistestausvetoisen ohjelmistokehitykseen tarkoitettu ohjelmistokehys. Concordion tarkoitus on helpottaa hyväksymistestien kirjoittamista, automatisointia, ja testien tulosten visualisoimista. Concordion käyttää määrittelydokumenteissaan XHTML-kieleltä (eXtensible Hypertext Markup Language). XHTML on HTML-kielen alakieli, joka täyttää XML-kielen muotovaatimukset. Concordionin käyttämät komennot on upotettu dokumentin elementtien attribuutteihin ja ne eivät nä, esimerkiksi selaimella avatussa tiedostossa. Concordion tuottamasta raportista tulee selkeästi esiin, mitä kyseessä olevalla testillä testattiin ja menikö testi läpi. Concordion tukee muun muassa taulukkomuotoista testiaineistoa, jolloin samaa toiminnallisuutta voidaan testata helposti ja selkeästi eri arvoilla.

*JMock*

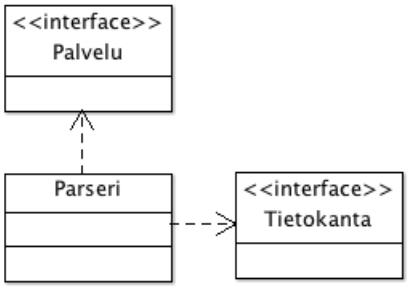
JMock on matkijakirjasto, jolla voidaan matkia luokkia ja rajapintojen kutsumista ja niiden palauttamia arvoja. JMockin avulla ohjelmistokehittäjä voi testata, että ulkoista järjestelmää kutsutaan koodissa oikeilla parametreilla ja esimerkiksi vain yhden kerran. JMockia käyttäessään ohjelmistokehittäjä määrittelee testissään testattavan metodin sisäisen toiminnan. Tällöin esimerkiksi lokituksen lisääminen voi aiheuttaa testin hajoamisen, vaikka metodin ulospäin näkyvä toiminnallisuus ei ole muuttunut. Tätä ongelmaa voidaan kiertää käyttämällä tynkäluokkia, mutta aina tämä ei ole mahdollista.

*Spring*

Spring on alkujaan ohjelmiston sisäisten riippuvuuksien hallintaan tarkoitettu sovelluskehys. Nykyään Spring pitää sisällään paljon erilaisia osia, kuten MVC-verkkosovelluskehyksen (Spring-MVC), järjestelmien integraatiokehyksen (Spring-Integration) ja tietoturvakehyksen (Spring-Security). Tässä projektissa käytetään Springin tietokantakehystä (Spring-Data), joka helpottaa tietokantakoodin testaamista. Kehys myös vähentää pakollisen niin kutsutun ”boilerplate”-koodin kirjoittamista. Boilerplate-koodi tarkoittaa koodia, jota tarvitaan ohjelman suorittamiseen, mutta joka ei sisällä ohjelman toimintalogiikan kannalta mitään tärkeää. Tietokantaan liittyvässä koodissa tällaistä esimerkiksi on yhteyden avaus ja sulkeminen.

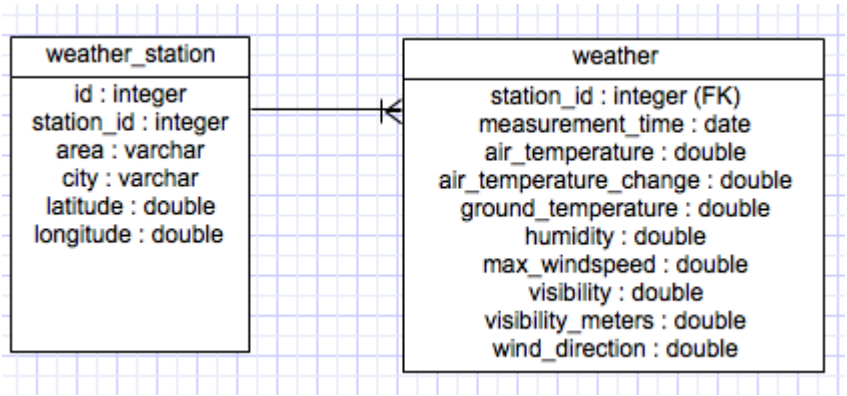
## 4.2 Arkkitehtuuri

Vaikka TDD:ssä ja ATDD:ssä testien tulisi ohjata ohjelmiston rakennetta, on yleensä hyvä ennen ohjelmoinnin aloittamista miettiä, mitä osia ohjelmisto tarvitsee toimiakseen. Tähän tarkoitukseen voidaan käyttää pelkkää lehtiötaulua, johon ohjelmiston osat voidaan piirtää tai voidaan käyttää esimerkki UML-kuvauskieltä (Unified Modeling Language). Kuvassa 7 on esitetty hyvin korkealla tasolla, mitä osia ohjelmaan kuuluu.




Kuva 7 Korkean tason kuva tiesääparserin osista.

Ohjelman osat ovat säätiedot tarjoava palvelu, tietokanta, johon sääasemien tiedot ja säätiedot tallennetaan ja parseri, joka käsittelee palvelulta tulleen vastauksen. Kuvan perusteella ohjelmistokehitys kannattaa aloittaa parserista, koska se on käyttäjä kahdelle muulle osalle ja näin saadaan ensimmäinen ja ainoa kutsuja molemmille osille. Ohjelman hakemat tiedot tallennetaan tietokantaan (Kuva 8).




Kuva 8 Tietokantakuvaus

Tietokanta koostuu weather_station- ja weather-tauluista. Taulut on sidottu toisiinsa palvelun luoman station_id:n avulla. Weather_station-taulussa on uniikkiavaimena station_id ja id-kenttien liitos. Weather-taulussa on vierasavaimena station_id, joka viittaa weather_station-taulun station_id-kenttään. Tässä ratkaisussa on vaarana, että jos palvelu palauttaa vanhan station_id:tä uudella id:llä, niin weather_station-tauluun luodaan uusi rivi. Tämän jälkeen ei enää voida luotettavasti sanoa, kummalle sääasemalle säätiedot kuuluvat. Tämä riski hyväksytään, koska projektissa on tarkoitus kerätä tietoa vain ly-

hyen aikaa, ja tiedon keruu voidaan aloittaa aina uudestaan.
Tietokannaksi olisi voitu valita myös jokin NoSQL-tietokanta. NoSQL-tietokannoilla tarkoitetaan tietokantoja, joissa tieto ei tallenneta tauluihin, vaan esimerkiksi avain-arvo-pareina. Projektiin valittiin kuitenkin perinteinen relaatiotietokanta, koska sen ominaisuudet ovat tunnettuja ja tallennettava tieto on hyvin rakenteellista eikä rakenteeseen ole näkyvissä muutoksia.

## 4.3 Käyttäjätarinan tekeminen

Taulukossa 2 on esitetty kaksi käyttäjätarinaa tehtävälle ohjelmalle ja niistä johdetut hyväksymistestitapaukset.

Taulukko 2 Projektin alkuperäiset hyväksymistarinat

| **Tarina** | **Testitapaukset** |
|---|---|
| Kun palvelua kutsutaan ja saadaan vastauksena yhden uuden sääaseman tiedot ja sää, niin luodaan yksi uusi sääasema ja tallennetaan sääaseman säätiedot . | - Uuden sääaseman tiedot on tallennettu.<br><br>- Uuden sääaseman säätiedot ovat tallennettu. |
| Kun palvelua kutsutaan ja saadaan vastauksena yhden vanhana sääaseman tiedot ja sää, niin tallennetaan vain sääaseman säätiedot . | - Vanhan sääaseman säätiedot ovat tallennettu. |

Oletetaan, että palvelu palauttaa kahden sääaseman tiedot ja niiden säätiedot. Toinen sääasema on jo olemassa ja toinen on uusi. Näin ei tarvita kahta hyväksimystestiä, vaan molemmat voidaan testata yhdellä hyväksymistestillä (taulukko 3).

Taulukko 3 Projektin hyväksymistarinat yhdistettynä

| **Tarina** | **Testitapaukset** |
|---|---|
| Kun ohjelma kutsuu palvelua ja saadaan vastaukset yhden uuden sääaseman tiedot ja sää, sekä yhden jo olemassaolevan sääaseman tiedot ja sää, niin luodaan yksi uusi sääasema ja molempien sääasemien säätiedot tallennetaan. | - Uuden sääaseman tiedot on tallennettu.<br><br>- Sääasemien säätiedot ovat tallennettu ja ne ovat sidottu oikeisiin sääasemiinsa. |

## 4.4 Tarinan automatisointi

Ohjelma käyttää kahta ulkoista järjestelmää: säätiedot tarjoavaa rajapintaa ja tietokantaa, johon käsitellyt säädetiedot tallennetaan. Hyväksymistestitapauksen automatisoinnissa pitää päättää, kutsutaanko testissä oikeaa rajapintaa ja tietokantaa vai korvataanko ne omilla tynkätoteutuksilla.

Kutsuttava rajapinta on puhtaasti ulkoinen riippuvuus. Ohjelmistolla ei ole mitään valtaa siihen, mitä rajapinta palauttaa tai vastaako se edes. Oikean rajapinnan kutsuminen vaatii myös aina toimivan verkkoyhteyden. Näiden asioiden perusteella hyväksymisteissä käytetään rajapinnan toteuttavaa tynkätoteutusta, joka palauttaa aina vakioviestin. Näin pystytään hallitsemaan testissä yksiselitteisesti, mitä rajapinta palauttaa. Samalla päästään verkkoyhteysvaatimuksesta.
Tietokanta on hieman toisenlainen riippuvuus. Tietokantaan testeissä pystytään vaikuttamaan siihen onko testin alussa tietokannan taulut tyhjät vai onko niissä rivejä. Hyväksymistesteissä käytetään pelkästään tietokoneen muistissa ajettavaa tietokantaa. Tietokoneen muistissa pyörivän tietokannan hyviä puolia ovat nopeus, ja se lakkaa olemasta, kun hyväksymistestit on ajettu.
Esimerkissä 4 on esitelty, miltä hyväksymistestitapaus näyttää Concordionin ymmärtämässä XHTML-muodossa.

```
1  <html xmlns:concordion="http://www.concordion.org/2007/concordion">
2  <link href="../concordion.css" rel="stylesheet" type="text/css" />
3  <head></head>
4  <body>
5      <h1>Yleiset tiedot</h1>
6      <p>
7          Ohjelma käyttää REST-palvelua osoitteessa
8          http://tieinfo.mustcode.fi/tieinfo/saa/paikkakunta/helsinki?json
9      </p>
10     <div class="test_case">
11                 <h3>Testi</h3>
12         <p>
13         Kun palvelua kutsutaan ja saadaan vastaukseksi
14         <span concordion:execute="#result = parse()"/> yhden
   uuden sääaseman tiedot ja sää,
15         sekä yhden jo olemassaolevan sääaseman tiedot jasää, niin luodaan yksi uusi sääasema ja
16         molempien sääasemien säätiedot tallennetaan.
17         <br/>
18         – Uuden sääaseman tiedot on tallennettu, jolloin tietokannasta löytyy
19         kaksi (<span concordion:assertEquals="#result.weatherStationCount">2</span>) sääasemaa.
20         <br/>
21         – Sääasemien säätiedot ovat tallennettu ja ne ovat
22         <span concordion:assertTrue="#result.rowsMatch"> sidottu oikeisiin sääasemiinsa</span>.
23         </p>
24     </div>
25 </body>
26 </html>
```

Esimerkki 4 Tiesääparserin hyväksymistesti kirjoitettu Concordionille.

Listauksessa näkyy, kuinka taulukkossa 3 määritellyt vaatimukset on kirjoitettu auki yhdeksi testitapaukseksi. Esimerkin 4 rivillä 13 kutsuttavan parse-metodin toteutus on kuvattu esimerkissä 5.

```
1      public ParserResult parse(){
2          RoadWeatherParser parser = new RoadWeatherParser(new StubWeatherService(),
3                                                           new StubWeatherServiceDAO());
4          parser.parseWeather("");
5          JdbcTemplate template = new JdbcTemplate(database);
6          int weatherStations = template.queryForInt(COUNT_WEATHER_STATION_ROWS);
7          int weatherRowsMatch = template.queryForInt(MATCH_WEATHER_ROWS_WITH_STATIONS);
8          ParserResult parserResult = new ParserResult();
9          parserResult.weatherStationCount = weatherStations;
10         parserResult.rowsMatch = (weatherRowsMatch == 2);
11         return parserResult;
12     }
13     class ParserResult{
14         public int weatherStationCount = 0;
15         public boolean rowsMatch = false;
16     }
```

Esimerkki 5 Ensimmäinen versio Concordionilla kirjoitetusta hyväksymistestistä.

Metodin ensimmäisellä rivillä luodaan RoadWeatherParser-olio, joka saa parametrikseen kaksi tynkätoteutusta rajapinnoista (StubWeatherService ja StubWeatherServiceDAO). Nämä rajapinnat ovat tässä ensimmäisessä versiossa vielä tyhjiä rajapintoja. Ne esittävät niitä kahta ulkoista palvelua, jotka tarvitaan, että parsimelle voidaan tehdä hyväksymistestit.

Rajapintojen tekeminen tässä vaiheessa voi vaikuttaa hieman oudolta. Mutta jotta ohjelman käyttämä palvelu voidaan korvata tynkätoteutuksella, joudutaan se antamaan jotenkin parserille. Samalla tehtiin tiedon tallentamisesta vastaava rajapinta. Tätä tapaa kutsutaan toiveajatteluksi (Gärtner 2012, s. 49). Toiveajattelun idea tässä konteksissa on, että hyväksymistestin kirjoittaja tekee tarvitsemansa rajapinnat ”tällainen olisi kiva” -tyylillä.

Näin hyväksymistestin kirjoittaja pääsee testin kirjoittamisessa eteenpäin eikä joudu miettimään hyväksymistestin kannalta epäolennaisia asioita, kuten ohjelman sisäisesti käyttämän rajapinnan totetutusta. Toiveajattelutavassa joudutaan luonnollisestikin korjaamaan hyväksymistestiä toteutuksen kehittyessä. Tämä ei ole ongelma, koska hyväksymistestiä on hyvin vaikea tehdä heti kerralla kuntoon, vaan testin sisäinen rakenne muuttuu ja kehittyy muun koodin mukana. Alussa riittää, että hyväksymistesti kääntyy ja näyttää punaista.

## 4.5 Parsimen testaus

Kuten kuvasta 7 näkee, sääparsimen ohjelmointi kannattaa aloittaa koodista, joka käsittelee palvelulta saadut tiedot ja antaa ne tietokantakoodille tallennettavaksi. Näin saadaan oikea käyttäjä niin palvelua kutsuvalle koodille kuin tiedon tallennuksesta vastaavalle koodille.

Ensimmäinen testi oli, että parsin kutsuu palvelua saadakseen säätiedot. Tätä varten jouduttiin luomaan palvelulle rajapintaluokka, jonka palvelu toteuttaa. Tiedon tallentamisesta vastaavalle luokalle luotiin myös tyhjä rajapintaluok-

ka, jotta testikoodissa voidaan testata, että parsin kutsuu tietokantakoodia. Rajapinnat on esitelty yhdessä esimerkissä 6.

```
1  /**
2  * Palvelun rajapinta
3  */
4  public interface WeatherService {
5          public String getWeatherInformation(String url);
6  }
7
8  /**
9  * Tiedon tallenuksen rajapinta
10 */
11 public interface WeatherServiceDAO {
12         public void saveWeatherInformation(List stationInformation);
13 }
```

Esimerkki 6 Ensimmäiset versiot ulkopuolisten järjestelmien rajapinnoista.

Rajapintojen suunnittelussa käytettiin ”toiveajattelua”, eli tällainen metodi sopisi tähän hyvin. Tämä ajattelutapa tulee esiin varsinkin WeatherServiceDAO-rajapintaluokassa, jossa on metodi saveWeatherInformation. Se vastaa sääasemien tietojen tallentamisesta. Tässä vaiheessa metodi saa parametrikseen pelkästään listan, jonka tyyppiä ei ole määritelty.

### 4.5.1 Ensimmäinen testi

Parseri toimii siten, että se kutsuu palvelua, jolta saa vastauksen. Vastauksen saatua se parsii sääasemien tiedot ja antaa ne listana tallennuksesta vastaavalle luokalle. Tästä saadaan ensimmäinen testi, joka testaa, että WeatherService- ja WeatherServiceDAO-rajapintaluokkien metodeja kutsutaan (esimerkki 7).

```
1  public void shouldConnectToServiceAndSaveInformation() {
2          Mockery mockery = new Mockery();
3          final WeatherService service = mockery.mock(WeatherService.class);
4          final WeatherServiceDAO dao = mockery.mock(WeatherServiceDAO.class);
5
6          mockery.checking(new Expectations(){{
7              oneOf(service).getWeatherInformation("");
8              oneOf(dao).saveWeatherInformation(with(any(List.class)));
9          }});
10         RoadWeatherParser parser = new RoadWeatherParser(service, dao);
11         parser.parseWeather("");
12         mockery.assertIsSatisfied();
13 }
```

Esimerkki 7 Ensimmäinen testi parserille.

Testin toisella rivillä luodaan ensin matkijaolio, jonka avulla luodaan rajapintaluokista oliot (rivit kolme ja neljä). Riviltä kuusi alkaen kerrotaan, miten matkittavien luokkien odotetaan käyttäytyvän. Esimerkiksi rivillä seitsemän kerrotaan, että service-olion metodia getWeatherInformation kutsutaan kerran tyhjällä merkkijonolla. Vastaavasti rivillä kahdeksan kerrotaan, että WeatherServiceDAO-luokan metodia saveWeatherInformation-metodia kutsutaan List-tyyppisellä oliolla.

Rivillä 10 luodaan itse parseri, jolle annetaan konstruktorissa parametriksi mainitut rajapintaluokkien ilmentymät. Säätietojen parsinnan suorittavaa metodia kutsutaan rivillä 11. Testin lopuksi rivillä 13 tarkistetaan matkijaoliolta, että kaikkia odotettuja metodeja kutsuttiin. Yksinkertaisin koodi, jolla tämä testi saadaan menemään läpi on esitelty esimerkki 8.

```
1 public void parseWeather(String url) {
2   this.weatherService.getWeatherInformation(url);
3   this.weatherServiceDAO.saveWeatherInformation(new ArrayList());
4 }
```

Esimerkki 8 Ensimmäinen versio parserista

Ensimmäinen versio parserin koodista ei tee mitään hyödyllistä. Se vain kutsuu palvelua saamallaan parametrilla. Tallentamisesta vastaavaa metodia kutsutaan tyhjällä listalla. Jotta parseri tekisi jotain hyödyllistä, niin sen pitää osata luoda palvelulta saamastaan vastauksesta lista olioita, jonka se voi lähettää edelleen tallennettavaksi WeatherServiceDAO-luokalle. Tätä varten testiä pitää muuttaa niin, että WeatherService palauttaa oikean palvelun mukaisen vastauksen ja WeatherServiceDAO:lta tarkistetaan tallennettava lista.

### 4.5.2 Toinen versio testistä

Matkitun WeatherService-rajapinnan pitää palauttaa oikean palvelun mukainen vastaus, jotta voidaan testata, että parseri osaa luoda vastauksesta oikeanlaisia olioita. Helpoimmin tämä onnistuu, kun kopioidaan palvelusta tuleva vastaus ja poistetaan siitä kaikkien muiden asemien tiedot, paitsi yhden (esimerkki 9).

```
1  public void shouldConnectToServiceAndSaveInformation() {
2          Mockery mockery = new Mockery();
3          final WeatherService service = mockery.mock(WeatherService.class);
4          final WeatherServiceDAO dao = mockery.mock(WeatherServiceDAO.class);
5
6          mockery.checking(new Expectations(){{
7              oneOf(service).getWeatherInformation("");
8              will(returnValue(getWeatherServiceResponse()))
9              oneOf(dao).saveWeatherInformation(with(any(List.class)));
10         }});
11         RoadWeatherParser parser = new RoadWeatherParser(service, dao);
12         parser.parseWeather("");
13         mockery.assertIsSatisfied();
14 }
15 private static final String getWeatherServiceResponse(){...}
```

Esimerkki 9 WeatherService-rajapinta palauttaa oikeanlaisen vastauksen.

Esimerkki 9 ei eroa paljoakaan esimerkissä 7 esitetystä koodista. Testikoodiin on lisätty paluuarvo (rivi 8) WeatherService-olion metodille getWeatherInformation. Paluuarvo saadaan getWeatherServiceResponse-metodilta, joka palauttaa JSON-muotoisen tekstin, joka sisältää yhden sääaseman tiedot. Teksti on jätetty esimerkistä pois, koska sen sisältö ei ole esimerkin kannalta tärkeää.

Jotta voidaan testata parsimen WeatherServiceDAO:lle antamaa listaa, joudutaan nykyisen testin matkijaolion luoma WeatherServiceDAO-olio korvaamaan tynkätoteutuksella (esimerkki 10).

```
1  public class WeatherServiceDAOStub implements WeatherServiceDAO{
2      private List<WeatherStationInformation> stationInformation;
3      public void saveWeatherInformation(List<WeatherStationInformation>
4                                          stationInformation) {
5          this.stationInformation = stationInformation;
6      }
7
8      public List<WeatherStationInformation> getInformation(){
9          return this.stationInformation;
10     }
11 }
```

Esimerkki 10 WeatherServiceDAO:n tynkätoteutus.

Tynkäluokka luonnollisesti toteuttaa WeatherServiceDAO-rajapinnan, jotta se voidaan antaa parsimille. Tynkäluokalla on muuttujana List-olio, joka pitää

sisällään WeatherStationInformation-olioita, jotta saveWeatherInformation-metodille annettu lista saadaan talteen. WeatherStationInformation oliot pitävät sisällään kaiken tallennettavan tiedon, mitä tietokantaan tullaan tallentamaan. Tynkäluokalla on lisäksi metodi, joka palauttaa saveWeatherInformation-metodille annetun listan, jotta testistä pääsee siihen käsiksi.

Testiä muutetaan niin, että matkijan luoma WeatherServiceDAO-olio korvataan WeatherServiceDAOStub-luokan ilmentymällä (esimerkki 11).

```
1  public void shouldConnectToServiceAndSaveInformation() {
2      Mockery mockery = new Mockery();
3      final WeatherService service = mockery.mock(WeatherService.class);
4      final WeatherServiceDAOStub dao = new WeatherServiceDAOStub();
5      mockery.checking(new Expectations(){{
6          oneOf(service).getWeatherInformation("");
7          will(returnValue(getWeatherServiceResponse()));
8      }});
9      RoadWeatherParser parser = new RoadWeatherParser(service, dao);
10     parser.parseWeather("");
11     assertEquals("Saved one weather information",1,dao.getInformation().size());
12     mockery.assertIsSatisfied();
13 }
```

Esimerkki 11 WeathServiceDAO:n tynkätoteutus testissä

Rivillä neljä luodaan WeatherServiceDAO-tyypin sijaan WeatherServiceDAOStub-tyyppinen olio. Tämä on tynkäluokka, jolla on metodi getInformation, jonka avulla saadaan saveWeatherInformation-metodille annettu lista. Matkijaluokan odotuksista on poistettu saveWeatherInformation-metodin kutsu tarpeettomana.
Rivillä 11 tarkistetaan, että WeatherServiceDAO:lle on annettu tallennettavaksi lista, jossa on yksi alkio. Koska TDD:n ideana on tehdä mahdollisimmian pieniä muutoksia, niin esimerkissä 12 on esitelty parserin koodi, joka läpäisee testikoodin.

```
1  public void parseWeather(String url) {
2      this.weatherService.getWeatherInformation(url);
3      ArrayList<WeatherStationInformation>stationInformations =
4                          new ArrayList<WeatherStationInformation>();
5      stationInformations.add(new WeatherStationInformation());
6      this.weatherServiceDAO.saveWeatherInformation(stationInformations);
7  }
```

Esimerkki 12 Toinen versio parserin koodista

Esimerkki 12 ei vieläkään tee mitään hyödyllistä, vaan kutsuu sääpalvelua ja antaa tietokannalle tallennettavaksi listan, jossa on yksi tyhjä olio. Seuraavaksi testiin lisätään tarkistuksia, että WeatherServiceDAO:lle annettu lista sisältää oikeilla säätiedoilla täytetyn WeatherStationInformation-olion. WeatherStationInformation-olion arvon tarkistamiseen käytetään JUnitin assertEquals-metodia, joka vertaa kahta arvoa toisiinsa ja antaa virheilmoituksen, jos arvot eivät täsmää (esimerkki 13).

```
1  public void shouldConnectToServiceAndSaveInformation() {
2          Mockery mockery = new Mockery();
3          final WeatherService service = mockery.mock(WeatherService.class);
4          final WeatherServiceDAOStub dao = new WeatherServiceDAOStub();
5          mockery.checking(new Expectations(){{
6              oneOf(service).getWeatherInformation("");
7              will(returnValue(getWeatherServiceResponse()));
8          }});
9          RoadWeatherParser parser = new RoadWeatherParser(service, dao);
10         parser.parseWeather("");
11         assertEquals("Saved one weather information",1,dao.getInformation().size());
12         WeatherStationInformation info = dao.getInformation().get(0);
13         assertEquals("Airtemp ",15.3,info.getAirTemperature(), 0);
14         assertEquals("Airtemp change",-0.1,info.getAirTemperatureChange(),0);
15         assertEquals("Avg windspeed",3.1, info.getAverageWindspeed(),0);
16         .
17         .
18         .
19         assertEquals("Wind direction",188,info.getWindDirection(),0);
20         mockery.assertIsSatisfied();
21 }
```

Esimerkki 13 Lisätty WeatherStationInformation-olion arvojen tarkistaminen.

Koska WeatherStationInformation-oliolla on 15 muuttujaa, niin esimerkissä ei ole listattu kaikkien arvojen tarkistusta. Sääpalvelun vastauksen käsittelyyn käytetään JSON-simple-nimistä kolmannen osapuolen kirjastoa. Valmiin kirjaston avulla säästetään aikaa ja vaivaa, eikä mahdollisesta jatkokehitystarpeesta tarvitse huolehtia. Toinen kolmannen osapuolen ohjelmistokirjasto tarvittiin sään mittausajan käsittelyyn. Sääpalvelu ilmoittaa ajan muodossa ”VVVV-KK-PPTHH:MM:SS±HH:MM”. Tälle muodolle Javassa ei löydy suoraan tukea. Ajankäsittelyyn valittiin Joda-time-niminen avoimen lähdekoodin kirjasto.

Parseriin tuli huomattavasti lisää koodia, mutta koodi on suurimmaksi osaksi

JSON-simple -kirjaston käyttöä, eikä se ole esimerkin kannalta tärkeää (esimerkki 14).

```
1  public void parseWeather(String url) {
2          String result = this.weatherService.getWeatherInformation(url);
3          List<WeatherStationInformation> stationInformations =
4                                      createWeatherInformationList(result);
5           this.weatherServiceDAO.saveWeatherInformation(stationInformations);
6
7  }
8  private List<WeatherStationInformation> createWeatherInformationList(
9                                                  String result) {...}
10 private Iterator getStations(String result) {...}
11 private Iterator getStationValues(Iterator stations){...}
12 private HashMap<WeatherStationParameter, String> parseStationInformation
13                                                  (Iterator stationValues){...}
14 private WeatherStationInformation createAndFillStationInformation(
15                          HashMap<WeatherStationParameter, String> values) {...}
```

Esimerkki 14 Kolmas versio parserista.

Parserin parseWeather-metodin rivimäärä on pysynyt samana ja säätietojen parsinta on piilotettu luokan yksityisiin metodeihin. Tallennettavan säätietolista luodaan createWeatherInformationList-metodissa, joka kutsuu getStations-, getStationsValues-, parseStationInformation- ja createAndFillStationInformation-metodeita mainitussa järjestyksessä. Tämä versio parserista on toimiva ja täyttää parserille asetetut odotukset.

### 4.5.3 Parserin ja testin refaktorointi

Parserin ja testin koodia refaktoroitiin koko ajan, jotta metodit pysyivät pieninä ja koodi helppolukuisena. Metodien pilkkomisen tarve tuli esiin varsinkin, kun RoadWeatherParser-luokkaan lisättiin JSON-palautteen käsittely. Esimerkissä 15 on esitetty, miltä RoadWeatherParser-luokan metodi parseWeather näytti ennen refaktorointia. Esimerkin koodi läpäisee esimerkin 11 testin ja on siis ulospäin näkyvälle toiminnallisuudeltaan täysin sama kuin esimerkki 14.

```
1  public void parseWeather(String url) {
2          String result = this.weatherService.getWeatherInformation(url);
3
4          Object resultObject = JSONValue.parse(result);
5          JSONArray resultArray = (JSONArray) resultObject;
6
```

```
7         Iterator stations = resultArray.iterator();
8         List<WeatherStationInformation> stationInformations =
9         new ArrayList<WeatherStationInformation>();
10        while(stations.hasNext()){
11            JSONObject station = (JSONObject) stations.next();
12            Iterator stationValues = station.entrySet().iterator();
13            HashMap<WeatherStationParameter, String> values =
14                        new HashMap<WeatherStationParameter, String>();
15            while(stationValues.hasNext()){
16                Map.Entry entry = (Entry) stationValues.next();
17                for(WeatherStationParameter parameter : WeatherStationParameter.values()){
18                    if(entry.getKey().equals(parameter.parameterName())){
19                        values.put(parameter, entry.getValue().toString());
20                     }
21                 }
22             }
23             WeatherStationInformation stationInformation =
24                                        WeatherStationInformation.build(values);
25            stationInformations.add(stationInformation);
26         }
27         this.weatherServiceDAO.saveWeatherInformation(stationInformations);
28    }
```

Esimerkki 15 Toimiva RoadWeatherParser ennen refaktorointia.

Esimerkissä 15 säätietojen arvojen muuttaminen tekstimuodosta on jo siirretty WeathStationInformation-luokan metodin build huoleksi (rivi 24). Koodin selkeyden takia parserWeather-metodi pilkottiin esimerkin 14 mukaiseksi. Esimerkissä 16 on viimeinen versio testistä ja sen tärkeimmistä metodeista.

```
1   public void setUp() throws Exception{
2         mockery = new Mockery();
3         service = mockery.mock(WeatherService.class);
4         dao = new WeatherServiceDAOStub();
5   }
6
7   public void shouldConnectToServiceAndSaveInformation() {
8          mockery.checking(new Expectations(){{
9              oneOf(service).getWeatherInformation("");
10             will(returnValue(getWeatherServiceResponse()));
11         }});
12         RoadWeatherParser parser = new RoadWeatherParser(service, dao);
13         parser.parseWeather("");
14         assertEquals("Saved one weather information",1,dao.getInformation().size());
15         WeatherStationInformation info = dao.getInformation().get(0);
16         assertEquals("Saved one weather information",1,dao.getInformation().size());
17         validateWeatherStationInfoValues(dao);
18         mockery.assertIsSatisfied();
19  }
20
21  private void validateWeatherStationInfoValues(final WeatherServiceDAOStub dao) {...}
```

Esimerkki 16 RoadWeatherParserin refaktoroitu yksikkötesti.

Testiä ei ole juurikaan tarvetta refaktoroida, vaan WeatherStationInformation-luokan arvojen tarkistus on siirrretty omaan metodiin (rivi 17). Mockery-, WeatherService- ja WeatherServiceDAOStub-olioista on tehty luokkamuuttujia, jotka alustetaan jokaista testiä varten uudelleen setUp-metodissa. Refaktoroinnin jälkeen testin rakenne on selkeä: asetetaan odotukset, suoritetaan testattava metodi ja tarkistetaan lopputulos.

## 4.6 Tietokantakoodin testaus

Tietokantakoodin testaamiseen käytetään avuksi Spring-ohjelmiston tarjoamia apuluokkia. Näiden avulla tietokantakoodin testaaminen on huomattavasti helpompaa.

### 4.6.1 Ensimmäinen testi

Ensimmäinen testi on varsin selkeä, sillä koko tietokantakoodin ainoa tarkoitus on tallentaa annetut tiedot tietokantaan. Ensimmäinen testi testaa, että tietokantakoodi kutsuu oikeanlaista SQL-kielen lausetta annetuilla arvoilla. Tämä testi testaa suoraan taulukossa 2 määriteltyä käyttäjätarinaa. Esimerkki 17 esittää ensimmäistä versiota testistä. Koska testit ovat yksikkötestejä, ne eivät mene tietokantaan tarkistamaan, että tietokantatauluihin on muodostunut oikeat rivit. Nämä testit testaavat vain, että metodeja kutsutaan oikeilla parametreilla.

```
1  public void newWeatherStationAndWeatherInfoAreInserted() throws Exception{
2          Mockery mockery = new Mockery();
3          mockery.setImposteriser(ClassImposteriser.INSTANCE);
4          final JdbcTemplate template = mockery.mock(JdbcTemplate.class, "weatherStationTemplate");
5          final WeatherServiceDAO weatherStationDao = new WeatherServiceDAOImpl(template);
6          mockery.checking(new Expectations(){{
7              oneOf(template).update("INSERT INTO weather_station " +
8              "(id, station_id, tsa_name, city, latitude, longitude) " +
9              "VALUES (?, ?, ?,?,?,?)",
10             new Object[]{ID, STATION_ID, TSA_NAME, CITY, LATITUDE, LONGITUDE});
11
12             oneOf(template).update("INSERT INTO weather " +
13              "(station_id, measurement_time, air_temperature, air_temperature_change, " +
14              "humidity,max_windspeed, visibility, visibility_meters, wind_direction) " +
15              "VALUES (?, ?, ?, ?,?,?,?, ?, ?, ?)",
16              new Object[]{STATION_ID,MEASUREMENT_TIME,AIR_TEMPERATURE,AIR_TEMPERATURE_CHANGE,
17              HUMIDITY,MAX_WINDSPEED, VISIBILITY, VISIBILITY_IN_METERS, WIND_DIRECTION
18              });
```

```
19          }});
20
21          List<WeatherStationInformation> infos = createListWithDummyWeatherStationInfos();
22          weatherStationDao.saveWeatherInformation(infos);
23
24          mockery.assertIsSatisfied();
25  }
```

Esimerkki 17 RoadWeatherDAO:n ensimmäinen yksikkötesti.

Rivillä 2 testissä luodaan matkijaluokan ilmentymä, jolle kerrotaan että matkittava luokka on Java-luokka, ei rajapinta (rivi 3). Riveillä 4 ja 5 luodaan matkittavien luokkien ilmentymät.
Riveillä 6 - 18 luodaan odotukset, että JdbcTemplate-olio tekee lisäyksen ensin weather_station-tauluun (rivi 7) ja sen jälkeen weather-tauluun. Isoilla kirjaimilla kirjoitetut muuttujat, kuten esimerkiksi ID, STATION_ID ja CITY ovat luokkamuuttujia, jotka edustavat tietokannan tauluihin laitettavia arvoja. Muuttujien arvoilla ei ole testin kannalta merkitystä.

Rivillä 21 luodaan lista, joka sisältää yhden WeatherStationInformation-olion, jolla on samat arvot muuttujissa mitä odotuksissa määriteltiin. Seuraavaksi kutsutaan tallennuksen tekevää koodia ja lopuksi tarkitetaan matkimelta, että odotukset täyttyivät (rivi 24). Testin läpäisevä koodi on vain muutaman rivin pituinen (esimerkki 18).

```
1   public void saveWeatherInformation(List<WeatherStationInformation>
2                                       stationInformations) {
3           for(WeatherStationInformation info : stationInformations){
4               template.update("INSERT INTO weather_station " +
5               "(id, station_id, tsa_name, city, latitude, longitude) VALUES " +
6               "(?, ?, ?,?,?,?)", createWeatherStationParameterArray(info));
7
8               template.update("INSERT INTO weather " +
9               "(station_id, measurement_time, air_temperature, " +
10              "air_temperature_change, humidity," +
11              "max_windspeed, visibility, visibility_meters, wind_direction) " +
12              "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)",createWeatherParameterArray(info));
13          }
14  }
15
16  private Object[] createWeatherStationParameterArray WeatherStationInformation info) {...}
17  private Object[] createWeatherParameterArray(WeatherStationInformation info){...}
```

Esimerkki 18 RoadWeatherDAO:n ensimmäinen versio.

Testattavassa metodissa käydään annettu lista alkioittain läpi, ja jokainen alkio lisätään tietokantaan (rivit 4 ja 8). Template-muuttuja on luokkamuuttuja, joka annetaan luokan konstruktorissa. Yksityiset apumetodit createWeatherStationParameterArray ja createWeatherParameterArray luovat WeatherSationInformation-olion muuttujista Object- tyyppiset taulukot, joissa on SQL-lauseilla syötettävät tiedot.
Toisin kuin WeatherServiceParserin ensimmäisissä versioissa (esimerkki 8), WeatherServiceDAO:n ensimmäinen versio tekee jo hyödyllistä ohjelman kannalta, eli syöttää tiedot tietokantaan.

### 4.6.2 Toinen testi

Ennen toisen testin tekemistä, suoritettiin koodin refaktorointi siten, että SQL-lauseet siirrettiin WeatherServiceDAO:n luokkamuuttujiksi. Näin SQL-lauseet ovat yhdessä paikassa ja niitä on tarvittaessa helpompi ylläpitää. Samalla Mockery-, JdbcTemplate- ja WeatherServiceDAO-luokista tehtiin luokkamuuttujat ja niiden alustus siirrettiin erillliseen setUp-metodiin, kuten RoadWeatherParser-luokan esimerkissä 16.

Toinen testi testaa tilannetta, jossa tietokannassa on jo olemassa oleva weather_station taulun rivi. Tässä tilanteessa WeatherServiceDAO:n tulee lisätä vain uusi weather-taulun rivi. WeatherServiceDAO joutuu siis ensin tarkistamaan, löytyykö sääasemalle jo riviä weather_station-taulusta. Jos löytyy, niin lisätään vain weather-taulun rivi (esimerkki 19).

```
1  public void existingWeatherStationOnlyWeatherInfoInserted() throws Exception{
2          mockery.checking(new Expectations(){{
3          oneOf(template).queryForInt("SELECT COUNT(*) FROM weather_station " +
4                                      "WHERE id = ? AND station_id = ?",
5          new Object[]{ID, STATION_ID});
6          will(returnValue(1));
7          never(template).update(weatherStationDao.INSERT_WEATHER_STATION,
8          new Object[]{ID, STATION_ID, TSA_NAME, CITY, LATITUDE, LONGITUDE});
9          oneOf(template).update(weatherStationDao.INSERT_WEATHER, new Object[]{
10          STATION_ID,MEASUREMENT_TIME,AIR_TEMPERATURE,AIR_TEMPERATURE_CHANGE,
11          HUMIDITY,MAX_WINDSPEED,VISIBILITY,VISIBILITY_IN_METERS,WIND_DIRECTION
12          });
13          }});
14          List<WeatherStationInformation> stations = createListWithDummyWeatherStationInfos();
15          weatherStationDao.saveWeatherInformation(stations);
```

```
16          mockery.assertIsSatisfied();
17  }
```

Esimerkki 19 RoadWeatherDAO:n toinen testi.

Testin odotuksiin on lisätty tietokantakysely, jolla lasketaan, montako riviä weather_station-taulusta löytyy tietyllä id:llä ja station_id:llä (rivi 3). Tässä testissä kyseinen kysely palauttaa arvon yksi, joka tarkoittaa, että taulusta löytyy sääasema. Rivillä seitsemän lisätään odotus, että weather_station-tauluun ei koskaan kohdistu lisäystä.

Loppuosa koodista on sama kuin edellisessä testissä, eli lisätään odotus, että weather-tauluun lisätään rivi (rivi 9), luodaan lista yhdellä täytetyllä WeatherStationInfo-oliolla, kutsutaan metodia ja tarkistetaan Mockery-oliolta, että kaikki odotukset täyttyivät.

WeatherStationDAO-luokkaan joudutaan lisäämään logiikka, joka tarkistaa ensin onko jo olemassa vastaava weather_station-taulun rivi, ja toimii sitten sen mukaan. Koodi on esitelty esimerkissä 20.

```
1  public void saveWeatherInformation(List<WeatherStationInformation>
2                                      stationInformations) {
3          for(WeatherStationInformation info : stationInformations){
4              if(isNewWeatherStation(info)){
5                  template.update(INSERT_WEATHER_STATION,
6                  createWeatherStationParameterArray(info));
7              }
8              template.update(INSERT_WEATHER, createWeatherParameterArray(info));
9          }
10 private boolean isNewWeatherStation(WeatherStationInformation info) {...}
```

Esimerkki 20 RoadWeatherDAO:n toinen versio.

WeatherServiceDAO:a on muutettu siten, että weather_station-taulun rivin lisäys on if-lauseen sisällä. If-lause kutsuu metodia isNewWeatherStation, joka tekee testissä määritellyn kyselyn tietokantaan ja tarkistaa, että tulos on nolla. If-lauseen lisäys rikkoo ensimmäisen testin, koska siinä ei ole määritelty weather_station-taululle tehtävää tarkistusta. Ensimmäisen testin tapauksessa tarkistuksen tulee palauttaa nolla, koska kyseessä on uusi weather_station-taulun rivi (esimerkki 21).

```
1  public void newWeatherStationAndWeatherInfoAreInserted() throws Exception{
2          mockery.checking(new Expectations(){{
3          oneOf(template).queryForInt(weatherStationDao.INSERT_WEATHER_STATION",
4          new Object[]{ID, STATION_ID});
5          will(returnValue(0));
6          oneOf(template).update(weatherStationDao.INSERT_WEATHER_STATION,
7          new Object[]{...});
8          oneOf(template).update(weatherStationDao.INSERT_WEATHER, new Object[]{...});
9           }});
10
11          List<WeatherStationInformation> infos = createListWithDummyWeatherStationInfos();
12          weatherStationDao.saveWeatherInformation(infos);
13          mockery.assertIsSatisfied();
14 }
15
16 public void existingWeatherStationOnlyWeatherInfoInserted() throws Exception{
17         mockery.checking(new Expectations(){{
18         oneOf(template).queryForInt(weatherStationDao.INSERT_WEATHER_STATION,
19         new Object[]{ID, STATION_ID});
20         will(returnValue(1));
21         never(template).update(weatherStationDao.INSERT_WEATHER_STATION,
22         new Object[]{...});
23         oneOf(template).update(weatherStationDao.INSERT_WEATHER, new Object[]{...});
24          }});
25          List<WeatherStationInformation> stations = createListWithDummyWeatherStationInfos();
26          weatherStationDao.saveWeatherInformation(stations);
27          mockery.assertIsSatisfied();
28 }
```

Esimerkki 21 RoadWeatherDAO korjattuineen testeineen.

Koodia refaktoroitiin samalla sen verran, että weather_station-tauluun tehtävä SQL-lause siirrettiin WeatherStationDAO-luokkaan muiden SQL-lauseiden mukaan. Testikoodissa on paljon toistoa, sillä ne eroavat toisistaan vain siinä, mitä weather_station-taulun tarkistus palauttaa ja tehdäänkö weather_station-tauluun lisäystä. Yhteisen koodin voisi refaktoroidan apumetodeihin, mutta henkilökohtainen mielipide on, että testeissä koodin toistoa voidaan sietää paremmin kuin tuotantokoodissa, jos se tekee testin tarkoituksen selkeämmäksi. Tilanne olisi toinen, jos testejä olisi enemmän.

## 4.7 Palvelunkutsujan testaus

Palvelua kutsuvan koodin testaaminen osoittautui oletettua hankalamamksi, koska osa Javan verkkoluokista on määritelty final-avainsanalla. Final-avainsanalla varustettuja luokkia tai metodeja ei voida periä tai ylikirjoittaa.

Tämän takia final-luokista ei voida tehdä normaalisti matkijaoliolla ilmentymiä.

Tämän ongelman kiertämiseen on työkaluja, jotka muokkaavat tavukoodia ennen kuin Javan virtuaalikone lataa luokan. Projektissa ei kuitenkaan käytetä näitä työkaluja, vaan ongelma kierrettiin toisella tapaa.

### 4.7.1 Ensimmäinen testi

Jotta palvelua kutsuvasta luokasta olisi hyötyä, niin sen pitää avata yhteys palveluun, lukea saamansa vastaus ja palauttaa saamansa vastaus kutsujalle. Yhteyden avauksen tarkistaminen ei onnistu ilman erillisiä työkaluja, koska Javan URL-luokka on final. Tämä ongelma kierrettiin niin, että WeatherServiceImpl-luokkaan tehtiin oletusnäkyvyydellä oleva metodi, joka huolehtii yhteyden avaamisesta ja palauttaa BufferedReader-luokan ilmentymän. BufferedReader-oliolta getWeatherInformation-metodi lukee palvelulta tulevan vastauksen.

Ratkaisun huonona puolena on, että nyt testi on sidottu itse toteutukseen. Esimerkissä 22 on esitelty ensimmäinen versio WeatherServiceImpl-luokan testeistä.

```
1  public void serviceReadsFromReaderAndReturnsResponse() throws Exception {
2          mockery.checking(new Expectations(){{
3           oneOf(reader).readLine();
4           will(returnValue(expectedResponse));
5           oneOf(reader).readLine();
6           will(returnValue(null));
7           oneOf(reader).close();
8           }});
9
10          String actualResponse = service.getWeatherInformation(SERVICE_URL);
11          assertEquals("Response was not what was expected", expectedResponse, actualResponse);
12          mockery.assertIsSatisfied();
13 }
```

Esimerkki 22 Palvelun kutsujan ensimmäinen testi.

Testin alussa alustetaan testissä tarvittavat luokkamuuttujat. BufferedReader-luokan ilmentymä reader on se olio, joka palauttaa palvelulta saatavan vastauksen rivi riviltä. WeatherServiceStubb-luokka on WeatherServiceImpl-luokan

aliluokka, joka ylikirjoittaa metodin, joka aukaisee yhteyden sääpalveluun. Ylikirjoitettu metodi palauttaa matkitun BufferedReader-luokan olion, jonka luokka saa konstruktorin argumenttina (rivi 5).

Seuraavana testikoodissa tulee oletusten asettaminen matkijaoliolle. Oletuksissa oletetaan, että reader-luokan readLine-metodia kutsutaan kerran, jolloin vastaukseksi annetaan merkkijono. Palautettavan merkkijonon sisällöllä ei ole testin kannalta merkitystä, vaan siitä on tehty luokkamuuttuja. ReadLine-metodia kutsutaan vielä kerran (rivi 10), jolloin palautetaan null-arvo. Null-arvo tarkoittaa tässä, että BufferedReader-oliolla ei ole enää vastaukseksi saatuja rivejä. Rivillä 12 varmistetaan, että avatut resurssit suljetaan. Testin tarkistuksissa testataan, että getWeatherInformation-metodilta saatava vastaus on sama, mikä määriteltiin oletuksissa ja tarkistetaan matkijaoliolta, että odotukset täyttyivät.Testin läpäisevän WeatherServiceImpl-luokan toteutus on yksinkertainen (esimerkki 23).

```
1  public String getWeatherInformation(String url) {
2          StringBuilder builder = new StringBuilder(512);
3          BufferedReader reader = null;
4          try{
5              reader = openReader(url);
6              String line = null;
7              while((line = reader.readLine()) != null){
8                  builder.append(line);
9              }
10         }
11         catch(IOException e){
12         System.err.println("IOExcpetion when opening URL '"+url+"'\n"
13                                         +e.getMessage());
14         }
15         finally{closeSource(reader); }
16         return builder.toString();
17 }
18 BufferedReader openReader(String url) throws IOException {...}
```

Esimerkki 23 Palvelun kutsujan ensimmäinen versio.

Metodin alussa alustetaan StringBuilder-tyyppinen olio, johon tallennetaan vastaukseksi saatavat rivit. Rivillä viisi kutsutaan yhteyden avaavaa metodia, joka palauttaa BufferedReader-olion. Oliolta pyydetään rivejä, niin kauan kunnes saadaan null-arvo (rivi 7). Saadut rivit tallennetaan, jokaisella lukukierroksella StringBuilder-olioon (rivi 8).

Rivillä 11 siepataan mahdollinen yhteyden avaamisesta johtuva poikkeus. Poikkeus lokitetaan virhevirtaan. Tämä tapa ei ole suositeltavaa, vaan pitäisi käyttää erillistä lokitusluokkaa, jota konfiguroidaan yhteisesti. Ottaen huomioon ohjelman tarkoitus ja se, että tämä on ainoa lokitusta vaativa paikka, niin ratkaisu on hyväksyttävä. Finally-lohkossa, joka suoritetaan aina, kutsutaan closeSource-metodia, joka huolehtii BufferedReader resurssin vapauttamisesta. Metodi tarkistaa, että BufferedReader-olio on alustettu ja huolehtii mahdollisesta sulkemisen aiheuttamasta poikkeuksesta.

### 4.7.2 Toinen testi

Toisessa testissä testataan, että mahdollinen yhteyden heittämä poikkeus ei aiheuta metodin ulkopuolella vaikutuksia, vaan metodi palauttaa tyhjän merkkijonon (esimerkki 24). Ennen testin tekoa, luokkamuuttujien alustamiset siirrettiin setUp-metodiin, kuten aikaisemmissakin testeissä.

```
1 public void IOExceptionReturnsEmptyResponse() throws Exception{
2         service.throwIOException(true);
3          String actual = service.getWeatherInformation(SERVICE_URL);
4          assertEquals("Malformed URL expected empty response", "", actual);
5 }
```

Esimerkki 24 Palvelun kutsujan toinen testi

Testin toisella rivillä WeatherServiceStub-olio asetetaan heittämään IOException-tyyppinen poikkeus, kun ylikirjoitettua openReader-metodia kutsutaan. Kolmannella rivillä kutsutaan itse metodia ja tarkistetaan, että metodi palauttaa tyhjän merkkijonon.

Tämän testin läpäisemiseks ei tarvinnut muuttaa WeatherServiceImpl-luokan toteutusta, vaan esimerkissä 23 esitetty toteutus läpäisee testin. Tätä testiä kirjoitettaessa huomattiin, että muissakaan testiluokissa ei testattu tilannetta, jossa metodit saavat joko tyhjän listan käsiteltäväkseen (WeatherServiceDAOImpl) tai tyhjän vastauksen (RoadWeatherParser). Nämä testit lisättiin, ja ne menivät suoraan läpi.

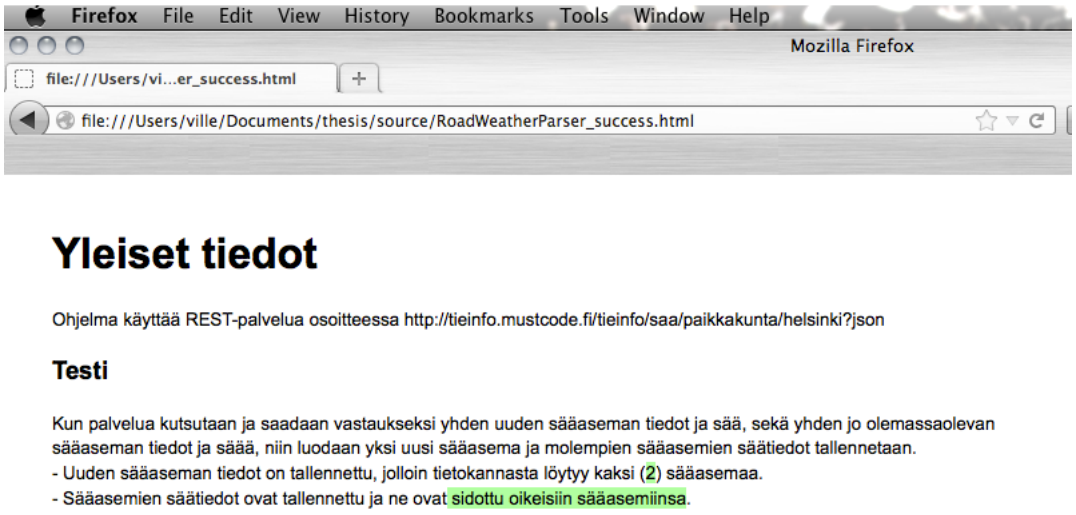
## 4.8 Hyväksymistestin läpäisy

Kun yksikkötestit on saatu tehtyä ohjelman eri osille, ajetaan hyväksymistesti, jotta nähdään toteuttaako ohjelma hyväksymismääritykset. Alkuperäistä testiä (esimerkki 4), joudutaan muokkaamaan erittäin vähän (esimerkki 25).

```
1  public ParserResult parse(){
2          JdbcTemplate template = new JdbcTemplate(database);
3          RoadWeatherParser parser = new RoadWeatherParser(
4                                          new StubWeatherService(),
5                                          new WeatherServiceDAOImpl(template));
6          parser.parseWeather("");
7          int weatherStations = template.queryForInt(COUNT_WEATHER_STATION_ROWS);
8          int weatherRowsMatch = template.queryForInt(MATCH_WEATHER_ROWS);
9          ParserResult parserResult = new ParserResult();
10         parserResult.weatherStationCount = weatherStations;
11         parserResult.rowsMatch = (weatherRowsMatch == 2);
12         return parserResult;
13
14 }
```

Esimerkki 25 Hyväksymistestin päivitetty versio.

Hyväksymistestissä vaihdetiin alkuperäisessä testissä käytetty StubWeatherxServiceDAO-luokka oikeaan tietokantaluokan toteutukseen (rivi 4). Testi ajettiin muutosten jälkeen uudelleen, jolloin se meni läpi ja saatiin kuvan 9 kaltainen testiraportti.




Kuva 9 Läpäisty hyväksymistesti.

## 4.9 Lähdekoodin tarkistus analysaattoreilla

Projektin koodi ajettiin Findbugs- ja PMD-koodianalysaattoreiden läpi koodin laadun tarkastamiseksi. Oletusasetuksilla Findbugs ei löytänyt ongelmia, jolloin asetukset asetettiin herkimmälle. Näillä asetuksilla Findbugs löysi 8 ongelmaa, koska sisäluokkia ei oltu määritelty staattiseksi.
PMD-analysaattori löysi projektista oletusasetuksilla 220 ongelmaa. Näistä ongelmista 2 oli prioriteeti 1- ja prioriteetti 2 -ongelmia, jotka ovat korkeimmat prioriteetit. Prioriteetti 1 -ongelma oli System.out.err-metodin käyttö lokittamiseen WeatherServiceImpl-luokassa. Prioriteetti 2 -ongelma oli IOExceptionReturnsEmptyResponse- testimetodin nimeäminen isolla alkukirjaimella.

Eniten virheitä oli prioriteetilla 3 (211 kappaletta). Suurin osa tämän kategorian ongelmista oli, että luokkamuuttujia pitäisi määritellä final-määritteellä. Myös liian pitkät muuttujien nimet ja että testimetodit heittävät Exception-poikkeuksen, joka on myös ongelma. Projektista ei siis löytynyt vakavia vikoja, sillä System.out.err-metodin käyttö on selitetty esimerkissä, ja testimetodien nimeämisessä selkeys on tärkeintä.

Testauslähtöisen ohjelmistokehityksen yksi idea on tuottaa kattava testiverkosto ohjelmistolle. Tässä projektissa tehdylle ohjelmalle saatiin testikattavuudeksi 96,1%, joka on todella hyvä tulos.

# 5 Yhteenveto

Insinöörityössä käytiin läpi miten ohjelmistoprojekti voidaan toteuttaa testauslähtöisesti. Teoriaosuudessa esiteltiin testauslähtöisen ohjelmistokehityksen vaiheittain ja miten ne toimivat yhdessä.

Insinöörityön projekti osuudessa tehtiin pieni ohjelma, joka hakee säätietoja verkkopalvelusta ja tallentaa ne tietokantaan, käyttäen teoriaosuudessa esiteltyjä vaiheita. Projektin aluksi määriteltiin käyttäjätarinoiden avulla, mitä ohjelman pitää tehdä. Näistä käyttäjätarinoista tehtiin ohjelmalle hyväksymistesti, joka automatisoitiin Concordion-testauskehyksellä. Hyväksysmistestin automatisoinnin jälkeen ohjelmaa alettiin kehittämään testivetoisesti osa osalta. Ohjelma koostui kolmesta osasta, joiden kehitys esiteltiin yksikkötestejä tekemällä kautta aina valmiiseen ohjelmaan asti, joka läpäisi alussa automatisoidun hyväksymistestin.

Esimerkkiprojektista saatiin lopputuloksena pieni ohjelma, jonka avulla voidaan kerätää säätietoja verkkopalvelusta myöhempää käyttöä varten. Vaikka esimerkkiprojektin ohjelma on varsin yksinkertainen, osoittaa se kuitenkin sen, että testauslähtöisellä ohjelmistokehityksellä voidaan toteuttaa ohjelmistoprojekteja. Samalla osoitettiin myös, että testauslähtöinen ohjelmistokehityksellä saadaan hyvä testikattavuus, sillä esimerkkiprojektin testikattavuus oli 96,1%.

Esimerkkiprojektin suurimmat haasteet olivat käyttäjätarinoiden kirjoittaminen ja hyväksymistestausympäristön luominen. Käyttäjätarinoiden kirjoittamisessa vaikeinta oli saada ne sellaiseen muotoon, että ne kertoivat mitä halutaan määrittelemättä toteutustapaa. Hyväksymistestausympäristön ongelma oli tietokannan luominen testin aluksi ja palauttaminen testin lopuksi alkutilaan. Ongelma ratkesi, kun löydettiin pelkästään muistissa pyörivä tietokanta, jolloin tietokannan siivoamisesta ei tarvinnut huolehtia. Vaikka testausympäristön pystyttäminen vei tässä esimerkkiprojektissa paljon aikaa, tehdään se

vain kerran projektin aikana. Toimivasta testausympäristöstä saatavat hyödyt ovat huomattavat, ja se kannattaa pystyttää, vaikka siihen kuluisi aikaa.

# Viitteet

*Findbugs* (2012), http://findbugs.sourceforge.net/. [Verkkoartikkeli, vierailtu 01.10.2012].

Fowler, M. (2012), 'Refaktorointi', http://martinfowler.com/refactoring/. [Verkkoartikkeli, vierailtu 04.04.2012].

Gärtner, M. (2012), *ATDD by example*, Addison Wesley.

Haikala, I. (2006), *Ohjelmistotuotanto*, Talentum.

*JUnit* (2012), http://www.junit.org/. [Verkkoartikkeli, vierailtu 02.10.2012].

Koskela, L. (2008), *Test Driven*, Manning.

Martin, R. C. (2011), *The clean coder*, Pearson Education Inc.

*PMD* (2012), http://pmd.sourceforge.net/. [Verkkoartikkeli, vierailtu 01.10.2012].

Steve Freeman, N. P. (2009), *Growing object-oriented software, guided by tests*, Addison Wesley.

*Vesiputousmalli* (2012), http://fi.wikipedia.org/wiki/Vesiputousmalli. [Verkkoartikkeli, vierailtu 29.02.2012].

*XUnit* (2012), http://en.wikipedia.org/wiki/XUnit. [Verkkoartikkeli, vierailtu 09.10.2012].

# A NameParser.java

```
1  package com.ville.friman.thesis;
2
3  public class NameParser {
4      private static final int FIRST_NAME = 0;
5      private static final int LAST_NAME = 1;
6
7      public FullName parse(String fullName) {
8          String[] splitNames = splitAndValidate(fullName);
9          return new FullName(splitNames[FIRST_NAME], splitNames[LAST_NAME]);
10     }
11
12     private String[] splitAndValidate(String fullName) {
13         checkForNullOrEmpty(fullName);
14         String[] splitNames = fullName.split(" ");
15         if(splitNames.length != 2){
16             throw new IllegalArgumentException("Parameter is too long or short");
17         }
18         return splitNames;
19     }
20
21     private void checkForNullOrEmpty(String fullName) {
22         if(fullName == null || fullName.trim().length() == 0){
23             throw new IllegalArgumentException("Parameter was null or empty");
24         }
25     }
26
27 }
```

# B Yksikkötestiluokka TestNameParser.java

```
1  package com.ville.friman.thesis;
2
3  import static org.junit.Assert.assertEquals;
4
5  import org.junit.Before;
6  import org.junit.Test;
7
8  public class TestNameParser {
9          private NameParser parser;
10
11         @Before
12         public void setUp() throws Exception {
13                 parser = new NameParser();
14         }
15
16         @Test
17         public void testParsingWithFirstNameLastName() throws Exception {
18                 FullName fullName = parser.parse("Matti Mainio");
19                 assertEquals("Matti", fullName.getFirstName());
20                 assertEquals("Mainio", fullName.getLastName());
21         }
22
23         @Test(expected = IllegalArgumentException.class)
24         public void nullValueShouldThrowException() throws Exception {
25                 parser.parse(null);
26         }
27
28         @Test(expected = IllegalArgumentException.class)
29         public void emptyStringShouldThrowException() throws Exception {
30                 parser.parse("");
31         }
32
33         @Test(expected = IllegalArgumentException.class)
34         public void whiteSpacesShouldThrowException() throws Exception{
35                 parser.parse("    ");
36         }
37
38         @Test(expected = IllegalArgumentException.class)
39         public void tooShortNameShouldThrowException() throws Exception{
40                 parser.parse("Matti");
41         }
42
43         @Test(expected = IllegalArgumentException.class)
44         public void tooLongNameShouldThrowException() throws Exception{
45                 parser.parse("Matti Jussi Mainio");
46         }
47 }
```

# C Concordion-testiluokka NameParserTest.java

```
1  package com.ville.friman.thesis;
2
3  import org.concordion.integration.junit3.ConcordionTestCase;
4
5  public class NameParserTest extends ConcordionTestCase{
6
7          public FullName split(String fullName){
8                  NameParser parser = new NameParser();
9                  return parser.parse(fullName);
10         }
11 }
```

# D Concordion-testin määrittely NameParser.html

```
1  <html xmlns:concordion="http://www.concordion.org/2007/concordion">
2      <body>
3          <div class="example">
4              <h3>Esimerkki</h3>
5              <p>
6                  Nimi
7                  <span concordion:execute="#result = split(#TEXT)">Matti Mainio</span>
8                  jakautuu etunimeen
9                  <span concordion:assertEquals="#result.firstName">Matti</span>
10                 ja sukunimeen
11                 <span concordion:assertEquals="#result.lastName">Mainio</span>.
12             </p>
13         </div>
14     </body>
15 </html>
```

# E Jaettua nimeä esittävä FullName.java

```
1  package com.ville.friman.thesis;
2
3  public class FullName {
4
5          private final String firstName;
6          private final String lastName;
7
8          public FullName(final String firstName, final String lastName){
9                  this.firstName = firstName;
10                 this.lastName = lastName;
11         }
12
13         public String getFirstName() {
14                 return firstName;
15         }
16
17         public String getLastName() {
18                 return lastName;
19         }
20 }
```

ß

# F TiesääParsijan XHTML-tiedosto

```
<html xmlns:concordion="http://www.concordion.org/2007/concordion">
<link href="../concordion.css" rel="stylesheet" type="text/css" />
<head></head>
<body>
    <h1>Yleiset tiedot</h1>
    <p>
        Ohjelma käyttää REST–palvelua osoitteessa
        http://tieinfo.mustcode.fi/tieinfo/saa/paikkakunta/helsinki?json
    </p>
    <div class="test_case">
                <h3>Testi</h3>
        <p>
        Kun palvelua kutsutaan ja saadaan vastaukseksi
        <span concordion:execute="#result = parse()"/> yhden  uuden sääaseman tiedot ja sää,
        sekä yhden jo olemassaolevan sääaseman tiedot jasää, niin luodaan yksi uusi sääasema ja
        molempien sääasemien säätiedot tallennetaan.
        <br/>
        – Uuden sääaseman tiedot on tallennettu, jolloin tietokannasta löytyy
        kaksi (<span concordion:assertEquals="#result.weatherStationCount">2</span>) sääasemaa.
        <br/>
        – Sääasemien säätiedot ovat tallennettu ja ne ovat
        <span concordion:assertTrue="#result.rowsMatch"> sidottu oikeisiin sääasemiinsa</span>.
        </p>
    </div>
</body>
</html>
```

# G TiesääParsijan hyväksymitesti

```
1  package com.villesoft.thesis.roadWeatherParser;
2
3  import org.concordion.integration.junit3.ConcordionTestCase;
4  import org.junit.After;
5  import org.junit.Before;
6
7  import org.springframework.jdbc.core.JdbcTemplate;
8  import org.springframework.jdbc.datasource.embedded.EmbeddedDatabase;
9  import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
10
11 import com.villesoft.thesis.roadWeatherParser.dao.WeatherServiceDAOImpl;
12 import com.villesoft.thesis.roadWeatherParser.service.WeatherService;
13
14 public class RoadWeatherParserTest extends ConcordionTestCase{
15         private static final String WEATHER_SQL = "sql/weather.sql";
16         private static final String WEATHER_STATION_SQL = "sql/weather_station.sql";
17
18     private static final String COUNT_WEATHER_STATION_ROWS =
19                 "SELECT COUNT(*) FROM weather_station";
20     private static final String MATCH_WEATHER_ROWS =
21                 "SELECT COUNT(weather.*) FROM weather JOIN weather_station ON " +
22                 "weather.station_id = weather_station.station_id";
23
24         private EmbeddedDatabase database;
25
26     @Before
27     public void setUp() throws Exception{
28         createDatabase();
29     }
30
31         private void createDatabase() {
32                 database = new EmbeddedDatabaseBuilder().
33                         addScript(WEATHER_STATION_SQL).
34                         addScript(WEATHER_SQL).
35                         build();
36         assertNotNull("Database creation failed", database);
37         }
38
39     @After
40     public void tearDown() throws Exception{
41         database.shutdown();
42     }
43
44         public ParserResult parse(){
45                 JdbcTemplate template = new JdbcTemplate(database);
46                 RoadWeatherParser parser = new RoadWeatherParser(
47                                 new StubWeatherService(),
48                                 new WeatherServiceDAOImpl(template));
49                 parser.parseWeather("");
50                 int weatherStations = template.queryForInt(COUNT_WEATHER_STATION_ROWS);
51                 int weatherRowsMatch = template.queryForInt(MATCH_WEATHER_ROWS);
52                 ParserResult parserResult = new ParserResult();
53                 parserResult.weatherStationCount = weatherStations;
54                 parserResult.rowsMatch = (weatherRowsMatch == 2);
55                 return parserResult;
56         }
```

```
57
58          class ParserResult{
59                  public int weatherStationCount = 0;
60                  public boolean rowsMatch = false;
61          }
62
63  private class StubWeatherService implements WeatherService{
64          public String getWeatherInformation(String url) {
65                  return "" +
66                  "[{\"class\":\"tieinfo.WeatherStation\",\"id\":1," +
67                  "\"airTemperature\":\"15.3\"," +
68                  "\"airTemperatureChange\":\"-0.1\",\"area\":\"Uusimaa\"," +
69                  "\"averageWindspeed\":\"3.1\"," +
70                  "\"bright\":\"1\",\"cameras\":[],\"cityName\":\"Helsinki\"," +
71                  "\"groundTemperature\":\"17.5\"," +
72                  "\"humidity\":\"97\",\"latitude\":60.159666666666666," +
73                  "\"longitude\":24.887883333333335," +
74                  "\"maxWindspeed\":\"5.3\"," +
75                  "\"measurementtime\":\"2012-09-12T12:58:00+03:00\"," +
76                  "\"roadSurfaceConditions1\":\"1\",\"roadSurfaceConditions2\":\"1\"," +
77                  "\"roadSurfaceTemperature\":\"18.1\"," +
78                  "\"roadSurfaceTemperatureChange\":\"0.1\"," +
79                  "\"stationId\":\"1002\",\"sunup\":\"1\"," +
80                  "\"tsaName\":\"kt51_Hki_Lapinlahti_R\"," +
81                  "\"visibility\":\"2\",\"visibilityMeters\":\"2000\"," +
82                  "\"warning1\":\"0\",\"warning2\":\"0\"," +
83                  "\"windDirection\":\"188\"},"+
84                  // Second weather station information starts
85                  "{\"class\":\"tieinfo.WeatherStation\",\"id\":3," +
86                  "\"airTemperature\":\"15.3\"," +
87                  "\"airTemperatureChange\":\"-0.1\",\"area\":\"Uusimaa\"," +
88                  "\"averageWindspeed\":\"3.1\"," +
89                  "\"bright\":\"1\",\"cameras\":[],\"cityName\":\"Helsinki\"," +
90                  "\"groundTemperature\":\"17.5\"," +
91                  "\"humidity\":\"97\",\"latitude\":60.159666666666666," +
92                  "\"longitude\":24.887883333333335," +
93                  "\"maxWindspeed\":\"5.3\",\"measurementtime\":" +
94                  "\"2012-09-12T12:58:00+03:00\"," +
95                  "\"roadSurfaceConditions1\":\"1\",\"roadSurfaceConditions2\":\"1\"," +
96                  "\"roadSurfaceTemperature\":\"18.1\"," +
97                  "\"roadSurfaceTemperatureChange\":\"0.1\"," +
98                  "\"stationId\":\"1003\",\"sunup\":\"1\"," +
99                  "\"tsaName\":\"kt51_Hki_Lapinlahti_R\"," +
100                 "\"visibility\":\"2\",\"visibilityMeters\":\"2000\",\"warning1\":\"0\"," +
101                 "\"warning2\":\"0\"," +
102                 "\"windDirection\":\"188\"}]";
103                 }
104         }
105 }
```

# H RoadWeatherParser.java ja testiluokka

```
1  package com.villesoft.thesis.roadWeatherParser;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5  import java.util.Iterator;
6
7  import java.util.List;
8  import java.util.Map;
9  import java.util.Map.Entry;
10
11 import org.json.simple.JSONArray;
12 import org.json.simple.JSONObject;
13 import org.json.simple.JSONValue;
14
15 import com.villesoft.thesis.roadWeatherParser.dao.WeatherServiceDAO;
16 import com.villesoft.thesis.roadWeatherParser.service.WeatherService;
17
18 public class RoadWeatherParser {
19
20         private WeatherService weatherService;
21         private WeatherServiceDAO weatherServiceDAO;
22
23         public RoadWeatherParser(WeatherService weatherService,
24                                  WeatherServiceDAO serviceDAO) {
25                 this.weatherService = weatherService;
26                 this.weatherServiceDAO = serviceDAO;
27         }
28
29         public void parseWeather(String url) {
30                 String result = this.weatherService.getWeatherInformation(url);
31                 List<WeatherStationInformation> stationInformations =
32                     createWeatherInformationList(result);
33                 this.weatherServiceDAO.saveWeatherInformation(stationInformations);
34
35         }
36
37         private List<WeatherStationInformation> createWeatherInformationList(
38                         String result) {
39                 List<WeatherStationInformation> stationInformations =
40                     new ArrayList<WeatherStationInformation>();
41                 Iterator stations = getStations(result);
42                 while(stations.hasNext()){
43                         Iterator stationValues = getStationValues(stations);
44                         HashMap<WeatherStationParameter, String> values =
45                             parseStationInformation(stationValues);
46                         WeatherStationInformation stationInformation =
47                             createAndFillStationInformation(values);
48
49                         stationInformations.add(stationInformation);
50                 }
51                 return stationInformations;
52         }
53
54         private Iterator getStations(String result) {
55                 Object resultObject = JSONValue.parse(result);
56                 JSONArray resultArray = (JSONArray) resultObject;
```

```
57                  return resultArray.iterator();
58          }
59
60          private Iterator getStationValues(Iterator stations) {
61                  JSONObject station = (JSONObject) stations.next();
62                  return station.entrySet().iterator();
63          }
64
65          private HashMap<WeatherStationParameter, String> parseStationInformation
66              (Iterator stationValues) {
67                  HashMap<WeatherStationParameter, String> values =
68                      new HashMap<WeatherStationParameter, String>();
69                  while(stationValues.hasNext()){
70                          Map.Entry entry = (Entry) stationValues.next();
71                          for(WeatherStationParameter parameter :
72                                  WeatherStationParameter.values()){
73                                  if(entry.getKey().equals(parameter.parameterName())){
74                                          values.put(parameter, entry.getValue().toString());
75                                  }
76                          }
77                  }
78                  return values;
79          }
80
81          private WeatherStationInformation createAndFillStationInformation(
82                          HashMap<WeatherStationParameter, String> values) {
83                  return WeatherStationInformation.build(values);
84          }
85  }
```

```
1   package com.villesoft.thesis.roadWeatherParser;
2   import static org.junit.Assert.assertEquals;
3   import java.util.Calendar;
4   import java.util.Date;
5   import java.util.List;
6   import org.junit.Before;
7   import org.junit.Test;
8   import org.jmock.Expectations;
9   import org.jmock.Mockery;
10  import com.villesoft.thesis.roadWeatherParser.service.WeatherService;
11
12  public class RoadWeatherParserUnitTest {
13
14          private Mockery mockery = new Mockery();
15          private static final String URL = "http://127.0.0.1";
16          private WeatherService weatherServiceMock;
17          private WeatherServiceDAOStub serviceDAOStub;
18
19          @Before
20          public void setUp(){
21                  weatherServiceMock = mockery.mock(WeatherService.class);
22                  serviceDAOStub = new WeatherServiceDAOStub();
23          }
24
25          @Test
26          public void parserShouldConnectToServiceAndPreserveResult() {
27                  RoadWeatherParser parser = new RoadWeatherParser(weatherServiceMock, serviceDAOStub);
28                  mockery.checking(new Expectations(){{
29                          oneOf(weatherServiceMock).getWeatherInformation(URL);
```

```
30                          will(returnValue(getWeatherServiceResponse()));
31                  }});
32
33                  parser.parseWeather(URL);
34                  List<WeatherStationInformation> preservedResult = serviceDAOStub.getInformation();
35                  assertEquals("Parser should have preserved one weatherstation's " +
36                                  "information", 1, preservedResult.size());
37                  validateParsedWeather(preservedResult);
38          }
39
40          private void validateParsedWeather(
41                          List<WeatherStationInformation> preservedResult) {
42                  WeatherStationInformation info = preservedResult.get(0);
43                  assertEquals("Airtemp ",15.3,info.getAirTemperature(), 0);
44                  assertEquals("Airtemp change",-0.1,info.getAirTemperatureChange(),0);
45                  assertEquals("Avg windspeed",3.1, info.getAverageWindspeed(),0);
46                  assertEquals("City","Helsinki",info.getCity());
47                  assertEquals("Ground temp",17.5, info.getGroundTemperature(),0);
48                  assertEquals("Humidity",97,info.getHumidity(),0);
49                  assertEquals("ID",1,info.getId());
50                  assertEquals("Latitude",60.159666666666666,info.getLatitude(),0);
51                  assertEquals("Longitude",24.887883333333335,info.getLongitude(),0);
52                  assertEquals("Max windspeed",5.3,info.getMaxWinspeed(),0);
53                  assertEquals("Measurement time",createMeasurementDate(), info.getMeasurementTime());
54                  assertEquals("Station id",1002,info.getStationId());
55                  assertEquals("Tsa name","kt51_Hki_Lapinlahti_R",info.getTsaName());
56                  assertEquals("Visibility",2,info.getVisibility(),0);
57                  assertEquals("Visibility meters",2000,info.getVisibilityInMeters(),0);
58                  assertEquals("Wind direction",188,info.getWindDirection(),0);
59          }
60          private Date createMeasurementDate() {
61                  Calendar calendar = Calendar.getInstance();
62                  calendar.set(Calendar.YEAR, 2012);
63                  calendar.set(Calendar.MONTH, 8);
64                  calendar.set(Calendar.DATE, 12);
65                  calendar.set(Calendar.HOUR_OF_DAY, 12);
66                  calendar.set(Calendar.MINUTE, 58);
67                  calendar.set(Calendar.SECOND, 0);
68                  calendar.set(Calendar.MILLISECOND, 0);
69                  return calendar.getTime();
70          }
71          private static final String getWeatherServiceResponse(){
72                  return "[{\"class\":\"tieinfo.WeatherStation\",\"id\":1,\"airTemperature\":\"15.3\","
73                                  "\"airTemperatureChange\":\"-0.1\",\"area\":\"Uusimaa\",\"averageWind
74                                  "\"bright\":\"1\",\"cameras\":[],\"cityName\":\"Helsinki\",\"groundTe
75                                  "\"humidity\":\"97\",\"latitude\":60.159666666666666,\"longitude\":24.
76                                  "\"maxWindspeed\":\"5.3\",\"measurementtime\":\"2012-09-12T12:58:00+0
77                                  "\"roadSurfaceConditions1\":\"1\",\"roadSurfaceConditions2\":\"1\","
78                                  "\"roadSurfaceTemperature\":\"18.1\",\"roadSurfaceTemperatureChange\"
79                                  "\"stationId\":\"1002\",\"sunup\":\"1\",\"tsaName\":\"kt51_Hki_Lapinl
80                                  "\"visibility\":\"2\",\"visibilityMeters\":\"2000\",\"warning1\":\"0\
81                                  "\"windDirection\":\"188\"}]";
82          }
83  }
```

# I WeatherParserDAO-rajapinta, WeatherParserDAOImpl.java ja testiluokka

```
1  package com.villesoft.thesis.roadWeatherParser.dao;
2
3  import java.util.List;
4
5  import com.villesoft.thesis.roadWeatherParser.WeatherStationInformation;
6
7  public interface WeatherServiceDAO {
8          public void saveWeatherInformation(List<WeatherStationInformation>
9                                                  stationInformation);
10
11 }
```

```
1  package com.villesoft.thesis.roadWeatherParser.dao;
2
3  import java.util.List;
4
5  import org.springframework.jdbc.core.JdbcTemplate;
6  import org.springframework.jdbc.core.support.JdbcDaoSupport;
7
8  import com.villesoft.thesis.roadWeatherParser.WeatherStationInformation;
9
10 public class WeatherServiceDAOImpl extends JdbcDaoSupport implements
11                 WeatherServiceDAO {
12
13         private final JdbcTemplate template;
14         static final String FIND_WEATHER_STATION = "SELECT COUNT(*) FROM weather_station " +
15                         "WHERE id = ? AND station_id = ?";
16
17         static final String INSERT_WEATHER_STATION = "INSERT INTO weather_station " +
18                         "(id, station_id, tsa_name, city, latitude, longitude) VALUES " +
19                         "(?, ?, ?,?,?,?)";
20         static final String INSERT_WEATHER = "INSERT INTO weather " +
21                         "(station_id, measurement_time, air_temperature, " +
22                         "air_temperature_change, humidity," +
23                         "max_windspeed, visibility, visibility_meters, wind_direction) " +
24                         "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)";
25
26         public WeatherServiceDAOImpl(JdbcTemplate template){
27                 this.template = template;
28         }
29
30
31         public void saveWeatherInformation(List<WeatherStationInformation>
32                                                 stationInformations) {
33                 for(WeatherStationInformation info : stationInformations){
34                         if(isNewWeatherStation(info)){
35                                 template.update(INSERT_WEATHER_STATION,
36                                         createWeatherStationParameterArray(info));
37                         }
38                         template.update(INSERT_WEATHER, createWeatherParameterArray(info));
39                 }
40         }
41
42         private boolean isNewWeatherStation(WeatherStationInformation info) {
```

```
43                  return template.queryForInt(FIND_WEATHER_STATION,
44                                  new Object[]{info.getId(), info.getStationId()}) == 0;
45          }
46
47          private Object[] createWeatherParameterArray(WeatherStationInformation info)
48          {
49                  return new Object[]{info.getStationId(), info.getMeasurementTime(),
50                                  info.getAirTemperature(), info.getAirTemperatureChange(),
51                                  info.getHumidity(),
52                                  info.getMaxWinspeed(), info.getVisibility(),
53                                  info.getVisibilityInMeters(), info.getWindDirection()};
54          }
55
56          private Object[] createWeatherStationParameterArray(
57                          WeatherStationInformation info) {
58                  return new Object[]{info.getId(), info.getStationId(), info.getTsaName(),
59                                  info.getCity(), info.getLatitude(), info.getLongitude()};
60          }
61
62  }
```

```
1   package com.villesoft.thesis.roadWeatherParser.dao;
2
3   import java.util.ArrayList;
4   import java.util.Date;
5   import java.util.List;
6
7   import org.jmock.Expectations;
8   import org.jmock.Mockery;
9   import org.jmock.lib.legacy.ClassImposteriser;
10
11  import org.junit.Before;
12  import org.junit.Test;
13  import org.springframework.jdbc.core.JdbcTemplate;
14
15  import com.villesoft.thesis.roadWeatherParser.WeatherStationInformation;
16
17  public class TestWeatherServiceDAOImpl{
18
19          private static final Integer ID = 10;
20          private static final Integer STATION_ID = 20;
21          private static final String TSA_NAME = "TSA";
22          private static final String CITY = "CITY";
23          private static final Double LONGITUDE = 30.0;
24          private static final Double LATITUDE = 40.0;
25          //Weather params
26          private static final Date MEASUREMENT_TIME = new Date();
27          private static final Double AIR_TEMPERATURE = 18.0;
28          private static final Double AIR_TEMPERATURE_CHANGE = 1.0;
29          private static final Double AVERAGE_WIND_SPEED = 8.9;
30          private static final Double AVERAGE_GROUND_TEMPERATURE = 7.0;
31          private static final Double HUMIDITY = 90.0;
32          private static final Double MAX_WINDSPEED = 15.0;
33          private static final Double VISIBILITY = 5.0;
34          private static final Double VISIBILITY_IN_METERS = 5000.0;
35          private static final Double WIND_DIRECTION =  180.0;
36
37
38
```

```
39          private  Mockery mockery;
40          private JdbcTemplate template ;
41          private WeatherServiceDAOImpl weatherStationDao;
42
43          @Before
44          public void setUp() throws Exception{
45                  mockery = new Mockery();
46                  mockery.setImposteriser(ClassImposteriser.INSTANCE);
47                  template = mockery.mock(JdbcTemplate.class, "weatherStationTemplate");
48                  weatherStationDao = new WeatherServiceDAOImpl(template);
49          }
50
51          @Test
52          public void newWeatherStationAndWeatherInfoAreInserted() throws Exception{
53                  mockery.checking(new Expectations(){{
54                          oneOf(template).queryForInt(weatherStationDao.FIND_WEATHER_STATION,
55                                          new Object[]{ID, STATION_ID});
56                          will(returnValue(0));
57                          oneOf(template).update(weatherStationDao.INSERT_WEATHER_STATION,
58                                          new Object[]{ID, STATION_ID, TSA_NAME, CITY, LATITUDE, LONGIT
59                          oneOf(template).update(weatherStationDao.INSERT_WEATHER, new Object[]{
60                                          STATION_ID,MEASUREMENT_TIME,AIR_TEMPERATURE,AIR_TEMPERATURE_C
61                                          MAX_WINDSPEED,VISIBILITY,VISIBILITY_IN_METERS,WIND_DIRECTION
62                          });
63                  }});
64
65                  List<WeatherStationInformation> stations = createListOfWeatherStationInfos();
66                  weatherStationDao.saveWeatherInformation(stations);
67                  mockery.assertIsSatisfied();
68          }
69
70          @Test
71          public void oldWeatherStationIsIgnoredOnlyWeatherInfoIsInserted() throws Exception{
72                  mockery.checking(new Expectations(){{
73                          oneOf(template).queryForInt(weatherStationDao.FIND_WEATHER_STATION,
74                                          new Object[]{ID, STATION_ID});
75                          will(returnValue(1));
76                          never(template).update(weatherStationDao.INSERT_WEATHER_STATION,
77                                          new Object[]{ID, STATION_ID, TSA_NAME, CITY, LATITUDE, LONGIT
78                          oneOf(template).update(weatherStationDao.INSERT_WEATHER, new Object[]{
79                                          STATION_ID,MEASUREMENT_TIME,AIR_TEMPERATURE,AIR_TEMPERATURE_C
80                                          MAX_WINDSPEED,VISIBILITY,VISIBILITY_IN_METERS,WIND_DIRECTION
81                          });
82                  }});
83                  List<WeatherStationInformation> stations = createListOfWeatherStationInfos();
84                  weatherStationDao.saveWeatherInformation(stations);
85                  mockery.assertIsSatisfied();
86          }
87
88          private List<WeatherStationInformation> createListOfWeatherStationInfos() {
89                  List<WeatherStationInformation> stations = new ArrayList<WeatherStationInformation>()
90                  WeatherStationInformation station = createDummyWeatherStationInformation();
91                  stations.add(station);
92                  return stations;
93          }
94
95          private WeatherStationInformation createDummyWeatherStationInformation() {
96                  return new WeatherStationInformation(AIR_TEMPERATURE, AIR_TEMPERATURE_CHANGE,
97                                  AVERAGE_WIND_SPEED, AVERAGE_GROUND_TEMPERATURE, HUMIDITY, ID, LONGITU
```

```
98                                  MEASUREMENT_TIME, STATION_ID, TSA_NAME, VISIBILITY, VISIBILITY_IN_MET
99              }
100 }
```

# J WeatherService-rajapinta, WeatherServiceImpl.java ja testiluokka

```
1  package com.villesoft.thesis.roadWeatherParser.service;
2
3
4  public interface WeatherService {
5          public String getWeatherInformation(String url);
6  }
```

```
1  package com.villesoft.thesis.roadWeatherParser.service;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.net.URL;
7  import java.net.URLConnection;
8
9  public class WeatherServiceImpl implements WeatherService{
10
11         public String getWeatherInformation(String url) {
12                 StringBuilder builder = new StringBuilder(512);
13                 BufferedReader reader = null;
14                 try{
15                         reader = openReader(url);
16                         String line = null;
17                         while((line = reader.readLine()) != null){
18                                 builder.append(line);
19                         }
20                 }
21                 catch(IOException e){
22                         System.err.println("IOExcpetion when opening URL '"+url+"'\n"
23                                                                 +e.getMessage());
24                 }
25                 finally{
26                         closeSource(reader);
27                 }
28                 return builder.toString();
29         }
30
31         private void closeSource(BufferedReader reader) {
32                 if(reader != null){
33                         try {
34                                 reader.close();
35                         } catch (IOException e) {
36                         }
37                 }
38         }
39
40         BufferedReader openReader(String url) throws IOException {
41                 URL serviceURL =  new URL(url);
42                 URLConnection connection = serviceURL.openConnection();
43                 BufferedReader reader = new BufferedReader(
44                                                 new InputStreamReader(connection.getInputStream()));
45                 return reader;
46         }
47
```

```
}
```

```
package com.villesoft.thesis.roadWeatherParser.service;

import static org.junit.Assert.assertEquals;

import java.io.BufferedReader;
import java.io.IOException;

import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.lib.legacy.ClassImposteriser;
import org.junit.Before;
import org.junit.Test;


public class TestWeatherServiceImpl {

        private static final String SERVICE_URL = "http://127.0.0.1/foobar" ;
        private static final String EXPECTED_RESPONSE = "EXPECTED_RESPONSE";
        private WeatherServiceStub service;
        private Mockery mockery;
        private BufferedReader reader;

        @Before
        public void setUp() throws Exception{
                mockery = new Mockery();
                mockery.setImposteriser(ClassImposteriser.INSTANCE);
                reader = mockery.mock(BufferedReader.class);
                service = new WeatherServiceStub(reader);
        }

        @Test
        public void serviceReadsFromReaderAndReturnsResponse() throws Exception {

                mockery.checking(new Expectations(){{
                        oneOf(reader).readLine();
                        will(returnValue(EXPECTED_RESPONSE));
                        oneOf(reader).readLine();
                        will(returnValue(null));
                        oneOf(reader).close();
                }});
                String actualResponse = service.getWeatherInformation(SERVICE_URL);
                assertEquals("Response was not what was expected", EXPECTED_RESPONSE, actualResponse)
                mockery.assertIsSatisfied();
        }

        @Test
        public void IOExceptionReturnsEmptyResponse() throws Exception{
                service.throwIOException(true);
                String actual = service.getWeatherInformation(SERVICE_URL);
                assertEquals("Malformed URL expected empty response", "", actual);
        }

        private static class WeatherServiceStub extends WeatherServiceImpl{

                private BufferedReader reader;
                private boolean throwIOE = false;

```

```
58                  public WeatherServiceStub(BufferedReader reader){
59                          this.reader = reader;
60                  }
61                  void throwIOException(boolean throwIOE){
62                          this.throwIOE = throwIOE;
63                  }
64                  BufferedReader openReader(String url) throws IOException {
65                          if(throwIOE){
66                                  throw new IOException("Throwing IOException in stub class");
67                          }
68                          return this.reader;
69                  }
70          }
71  }
```

# K Tietokannan taulukuvaukset

```
CREATE TABLE weather_station(
        id INTEGER,
        station_id INTEGER,
        tsa_name VARCHAR(255),
        city VARCHAR(255),
        latitude DOUBLE,
        longitude DOUBLE,
        UNIQUE (station_id,id)
);
```

```
CREATE TABLE weather(
        station_id INTEGER,
        measurement_time DATE,
        air_temperature DOUBLE,
        air_temperature_change DOUBLE,
        ground_temperature DOUBLE,
        humidity DOUBLE,
        max_windspeed DOUBLE,
        visibility DOUBLE,
        visibility_meters DOUBLE,
        wind_direction DOUBLE
);
```